DUDLEY KNOX LIBRARY
NAVAL POSTGRADUATE SCHOOL
MONTEREY, CALIF 93940

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

NPS MICRO-COBOL
an Implementation of
a subset of ANSI-COBOL
for a Microcomputer System

by

Hal R. Powell

June 1980

Thesis Advisor: M. S. Moranville

Approved for public release; distribution unlimited.

T195480

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER	2. GOVT ACCESSION NO. / 3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle): NPS MICRO-COBOL an Implementation of a subset of ANSI-COBOL for a Microcomputer System	5. TYPE OF REPORT & PERIOD COVERED: Master's Thesis; June 1980
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): Hal R. Powell	8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Naval Postgraduate School, Monterey, California 93940	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS: Naval Postgraduate School, Monterey, Californis 93940	12. REPORT DATE: June 1980
	13. NUMBER OF PAGES: # 336
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)	15. SECURITY CLASS. (of this report): Unclassified
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

NPS MICRO-COBOL
Navy Standard HYPO-COBOL
Microcomputers
Compilers
COBOL

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

A compiler for a subset of the Automated Data Processing Equipment Selection Office (ADPESO), HYPO-COBOL, has been implemented on a microcomputer. The implementation provides nucleus level constructs, interprogram communications, and file options from the ANSI COBOL package along with the PERFORM UNTIL, PERFORM VARYING and an enhanced version of the IF-THEN-ELSE construct that includes nesting and multiple program statements for both the "THEN" and "ELSE" clauses. These additional constructs from level two of ANSI

UN

COBOL provide for more flexibility and increased structural control. The language was implemented through a compiler and run-time package executing under the CP/M operating system of a Z-80 or an 8080 microcomputer-based system. Both the compiler and interpreter can be executed in 20K bytes of main memory. Aprogram consisting of 5K bytes of symbol table entries can be supported on this size machine. Modification of the compiler and interpreter programs can be accomplished to take advantage of larger machines. The programs that make up the compiler and interpreter package require 50K bytes of disk storage.

Approved for public release; distribution unlimited.

NPS MICRO-COBOL
an Implemention of
a subset of ANSI-COBOL
for a Microcomputer System

by

Hal R. Powell
Lieutenant, United States Navy
B.S., San Jose State University, 1973

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 1980

P7633
C.1

DUDLEY KNOX LIBRARY
NAVAL POSTGRADUATE SCHOOL
MONTEREY, CALIF 93940

ABSTRACT

A compiler for a subset of the Automated Data Processing Equipment Selection Office (ADPESO), HYPO-COBOL, has been implemented on a microcomputer. The implementation provides nucleus level constructs, interprogram communications, and file options from the ANSI COBOL package along with the PERFORM UNTIL, PERFORM VARYING and an enhanced version of the IF-THEN-ELSE construct that includes nesting and multiple program statements for both the "THEN" and "ELSE" clauses. These additional constructs from level two of ANSI COBOL provide for more flexibility and increased structural control. The language was implemented through a compiler and run-time package executing under the CP/M operating system of a Z-80 or an 8080 microcomputer-based system. Both the compiler and interpreter can be executed in 20K bytes of main memory. A program consisting of 5K bytes of symbol table entries can be supported on this size machine. Modification of the compiler and interpreter programs can be accomplished to take advantage of larger machines. The programs that make up the compiler and interpreter package require 50K bytes of disk storage.

TABLE OF CONTENTS

I. INTRODUCTION 8

A. BACKGROUND 8

B. OPERATING ENVIRONMENT 10

C. GOALS AND OBJECTIVES 10

D. PROBLEM DEFINITION 11

E. PROBLEM SOLUTION 11

F. SYSTEM OVERVIEW 13

II. NPS MICRO-COBOL COMPILER 16

A. GENERAL DESCRIPTION 16

B. SYMBOL TABLE 16

1. Numeric Values 20

2. Numeric Edit 20

3. Alpha or Alphanumeric 25

4. Alpha Edit 25

5. Tables 27

6. Labels 27

7. Files 30

8. Records 30

C. COMPILER MODULE "PART ONE" 33

1. Purpose 33

2. Control Actions 33

3. Symbol Table Entries 37

4. Intermediate Code Generation 38

5. Parser Actions 39

D. INTERFACE ACTIONS 49
E. COMPILER MODULE "PART TWO" 51
1. Purpose 51
2. Control Actions 51
3. Symbol Table Entries 51
4. Intermediate Code Generation 51
5. Parser Actions 55
III. NPS MICRO-COBOL INTERPRETER 68
A. GENERAL DESCRIPTION 68
B. MEMORY ORGANIZATION 69
C. INTERPRETER INTERFACE 72
D. PSEUDO-MACHINE INSTRUCTIONS 80
1. Format 80
2. Arithmetic Operations 80
3. Branching 81
4. Moves 85
5. Input-Output 88
6. Subroutine Instructions 91
7. Special Instructions 92
IV. SYSTEM DEBUGGING METHODS AND TOOLS 95
A. DEBUGGING METHODOLOGY 95
B. INTERACTIVE TOOLS 96
C. CROSS REFERENCE LISTINGS 96
D. VALIDATION TESTS 97
V. CONCLUSIONS AND RECOMMENDATIONS 98
APPENDIX A-NPS MICRO-COBOL USER'S MANUAL 101

APPENDIX B-LIST OF MICRO-COBOL RESERVED WORDS 157
APPENDIX C-MICRO-COBOL EXECUTION PROCEDURES 159
APPENDIX D-PART ONE AND PART TWO INTERNAL DATA STRUCTURES
AND SIGNIFICANT VARIABLES 165
APPENDIX E-MACHINE DEPENDENT VARIABLES 175
APPENDIX F-MICRO-COBOL PARSE TABLE GENERATION 177
APPENDIX G-LIST OF INOPERATIVE CONSTRUCTS 179
APPENDIX H-IBM TO MICROCOMPUTER TRANSFER PROCEDURES 180
APPENDIX I-DEBUGGING NPS MICRO-COBOL USING SID 182
COMPUTER LISTINGS 184
PART ONE 184
PART TWO 226
INTERP 270
READER 309
BUILD 311
INTRDR 320
DECODE 321
GRAMMER 327
PART ONE 327
PART TWO 330
LIST OF REFERENCES 334
INITIAL DISTRIBUTION 336

I. INTRODUCTION

A. BACKGROUND

The NPS MICRO-COBOL Compiler/Interpreter was initially (1976) [3] developed to demonstrate that it was feasible to implement a COBOL compiler on a microcomputer. It was known that the COBOL language used would have to be a subset of ANSI COBOL because of the restriction imposed by the size of a microcomputer memory. A subset of ANSI COBOL, specifically the Navy's Automated Data Processing Equipment Selection Office (ADPESO) HYPO-COBOL [4], was selected as the basis for the implementation. Additional motivation was provided by the DOD requirement that all computers used in a non-tactical environment be capable of executing COBOL programs.

The previous work was directed toward six major areas: 1.) selecting a suitable COBOL subset to operate on, 2.) developing the associated grammar for the language, 3.) determining what type of compiler to design, 4.) designing and coding the compiler, 5.) designing and coding the interpreter, and 6.) testing and debugging of the storage allocation and symbol table entries of the compiler.

The choice of a suitable language was originally based on HYPO-COBOL, since this is a Department of the Navy approved subset of COBOL, designed to place minimal

requirements on a system for compiler support. Where possible, short constructs were used in the place of longer ones. Where more than one reserved word served the same function in COBOL the shortest form was used. There is no optional verbage in the language, and no duplicate constructs perform the same function. Limits were placed on all statements that had a variable input format so that all statements had a fixed maximum length. Where possible, such constructs were removed completely from the language. In addition, user defined identifier names were limited to twelve characters to reduce symbol table storage requirements.

Rather than include the standard levels of implementation for all of the modules in HYPO-COBOL, constructs were included only as required. In addition to low level constructs, THE PERFORM UNTIL was included to allow better program structure. Further justification for the manner of subsetting and a highly detailed description of each element of the language is contained in the HYPO-COBOL language specifications reference 3.

The grammar for the MICRO-COBOL language was defined as LALR(1). The compiler design was based on a table-driven parser for the LALR(1) grammar. The algorithm used to develope the parse tables for the compiler was developed by W. R. Lalonge [20].

The basic design and coding of the compiler and

interpreter was completed prior to the current thesis work by Scott Allan Craig [3]. Modification to the original thesis work was conducted by Phil Mylet [18]. Initial testing and debugging of Part One was conducted by Jim Farlee and Michael Rice[9].

B. OPERATING ENVIRONMENT

The NPS MICRO-COBOL compiler and interpreter are designed to run under the CP/M operating system on an 8080 or Z80 based microcomputer with at least 20K bytes of main memory. The compiler programs are designed to use no more than 14K bytes of main memory, while the interpreter program uses approximately 12K bytes. The compiler and interpreter require 50K bytes of disk storage for the programs that make up the compiler/interpreter package. For information on creating MICRO-COBOL source programs and CP/M see references 5 and 6.

C. GOALS AND OBJECTIVES

The major goals of this work were 1.) Modify the existing compiler to allow use of the ADPESO validation test programs, 2.) Correct all known errors as outlined by Farlee and Rice[18], 3.) Implement all constructs not previously implemented, 4.) Verify that NPS MICRO-COBOL met HYPO-COBOL standards, and 5.) Extend the existing compiler/interpreter

package with some of the more frequently used high level COBOL constructs.

In addition to the above goals, it was considered beneficial to update and incorporate all previous documentation into the present NPS MICRO-COBOL compiler/interpreter documentation. This documentation is included in this thesis.

D. PROBLEM DEFINITION

For software performance assessment, a series of simple COBOL source programs and the Navy ADPESO HYPO-COBOL [4] validation test programs (HCCVS) were compiled and execution was attempted. Initial results of the ADPESO validation test programs produced over 400 compile and run time errors. Some of the errors were known previously as outlined in the previous thesis work by Farley and Rice[9]. The elimination of these problems plus the goals outlined above formed the foundation for this thesis.

E. PROBLEM SOLUTION

The ADPESO validation test programs could not be used for testing the compiler/interpreter until three areas were corrected. 1.) File I/O was inadequate to generate usable intermediate code, 2.) the IF-THEN-ELSE construct would not allow multiple statements to be performed, and 3.) the Move

Numeric Edited command was not implemented. The file I/O problem was corrected by Doug Loskot[15] as a class project early in this thesis effort. A new IF-THEN-ELSE construct allowing the use of multiple statements in both the "THEN" and "ELSE" clauses was implemented by Robert Hartel and Doug Stowers[19] as another class project. Implementation of the Move Numeric Edited command was completed by the author early in the thesis effort and allowed the validation test programs to be used for testing.

Once the validation programs could be compiled and executed, testing and debugging continued at a more rapid pace. All the errors exposed by the test programs as well as the known errors outlined in Appendix G of Farlee and Rice[9] were corrected, with the exception of the tests dealing with the Interprogram Communication Module.

The grammer in Part Two of the compiler was not constructed to allow the name of a called program to be stored. This required a change to the existing grammar. In addition to modifing the grammar for subroutine calls, a change to allow nesting IF-THEN-ELSE, NEXT SENTENCE option, the PERFORM VARYING verb, the COMPUTE verb and the logical operators "AND" and "OR" were defined in the grammar.

The grammar change was implemented in two steps. First the IF-THEN-ELSE statement, which included nesting and an END-IF clause, and the PERFORM VARYING statement was implemented as a class project by Carol Cagle[2]. The

present grammer is the result of the second change and includes the COMPUTE verb, logical operators, GIVING clause for the arithemetic operators and the change that enabled implementation of the Interprogram Communications module. In it's present form all of the specifications of HYPO-COBOL are met or exceeded. In addition to the constructs previously mentioned the new grammar will be able make the environment division optional, handle null paragraphs (paragraphs with no statements) and multiple open, close, display, add, and subtract statements as well as multi-dimensional tables. Appendix G contains a list of constructs that have been defined in the grammar but not yet implemented.

F. SYSTEM OVERVIEW

NPS MICRO-COBOL is a compiler/interpreter package. The compiler consists of three modules that combine to produce two files. The first file is an intermediate code file and the second is a list file containing any compilation errors and the line that caused the error. The first and second modules are combined together to form a module called COBOL.COM. The command COBOL <file name> initiates the compilation sequence. The first module (PART I) opens the input file, list file and code file, moves the second module, READER, to high memory for later use, and then

starts compiling the input file through the word PROCEDURE in the sentence PROCEDURE DIVISION. The symbol table is built starting at a storage location just above PART I and can use all available memory up to the base of the READER routine previously moved to high memory. After PPOCEDURE is parsed control is transfered to the READER routine which then copies the third module (PART II), into memory over PART I. Compilation continues to the end of the input file using the symbol table constructed from PART I. The symbol table can be added to by PART II up to and including the area previously used by the READER routine as READER is no longer needed. This scheme allows the use of all available free memory for the symbol table. At the end of the input file all files are closed and the compilation process is complete.

Error recovery/management is accomplished using the ad hoc panic mode technique discussed in Aho and Ullman [1]. Errors are annouced to the user by a two letter code. The user is required to look up the meanings of these codes in order to understand the full significance of each error but it was felt that this technique was necessary to keep the size of the compiler/interpreter package to a minimum.

The command EXEC <file name> causes the load routine BUILD to be loaded into memory. The BUILD routine opens the intermediate file created by the first phase and sets up the core image of the pseudo machine. Control transfers to the

INTRDR routine (Interpreter Reader) which reads the third module CINTERP into memory. This is the interpreter and once loaded control is passed to it and program execution begins.

II. NPS MICRO-COBOL COMPILER

A. GENERAL DESCRIPTION

The MICRO-COBOL compiler is a one pass compiler that scans and parses MICRO-COBOL source programs, and generates intermediate code (pseudo-instructions) for the interpreter (pseudo-machine). The scanner design is similar to most other scanner implementations. The parser is an LALR(1) table-driven design, implemented in the PLM80 programming language [10]. The parse tables, as stated before, were generated using an algorithm developed at the University of Toronto [20].

The compiler reads the source program from a disk file, extracts the needed information for the symbol table and writes pseudo-instructions to an intermediate code file. To accomplish this function, the compiler consists of three modules: PART ONE, READER, and PART TWO.

B. SYMBOL TABLE

The symbol table is the key data structure in the compiler. Information concerning identifiers, files, and records specified in the DATA DIVISION of the MICRO-COBOL source program is stored in the symbol table, along with labels specified in the PROCEDURE DIVISION.

The symbol table structure consists of: 1.) a sixty-four

address hash table, 2.) a fixed length field of fourteen bytes for each symbol table entry, and 3.) a variable length field to hold the name of each identifier. Since each identifier name is limited to fifteen ASCII characters the symbol table entry for identifiers can vary in length from fourteen to twenty-nine bytes. The bytes of each symbol table entry are grouped into various fields of either one or two bytes depending on the storage requirements. The fourteen bytes of the fixed length field entry are numbered from zero to thirteen and the variable length field begins with byte fourteen. In referencing a specific field a byte index with a value from zero to fourteen is utilized.

The symbol table entry for a single identifier could contain up to nine different attributes of that identifier, although not all identifiers required the full range of attributes. The various fields in the symbol table contained different information depending on whether, for example, an identifier was a numeric or alphanumeric type. Four of the fields contained the same information for all identifiers. These fields were: 1.) field zero (bytes zero and one) contained the collision link, 2.) field one (byte two) contained the type of the identifier, 3.) field two (byte three) contained the length of the identifier name, and 4.) field thirteen (byte fourteen) was the beginning of the ASCII character representation for the identifier name. It should be noted that an identifier of type FILLER would not

have a name associated with it, so field two would contain a zero and field fourteen would not exist.

Entry into the symbol table is accomplished by using a HASH function on the ASCII character representation of the identifier name. This function generates an even number between zero and 126. The number is used as an index into the hash table by specifying an offset from the base of the hash table. The hash table can hold sixty-four uniquely determined address references to identifiers. The hash table entry associated with each index reference heads a linked list of identifiers with the same HASH function value. The linked list structure provides for additional identifier storage and therefore the number of unique identifiers is not limited by the sixty-four index values generated by the HASH function. A zero entry in the hash table indicates that there is no identifier with that HASH function value. In tracing through the linked list of identifiers the most recently declared variable appears at the end of the list. See figure [II-1] for an example of the computation of a hash value. See figure [II-2] for an example of the hash table indexing and linking of hash values.

HASH VALUE COMPUTATION

HASH Function value: sum of identifier ASCII characters logically and with 3FH then shifted left (SHL) one bit.

HASHBASE = 2000H

H.F.(AB) = HASHBASE + SHL(((41H + 42H) AND 3FH),1) = 2006H

H.F.(BA) = HASHBASE + SHL(((42H + 41H) AND 3FH),1) = 2006H

FIGURE II-1

HASH TABLE, SYMBOL TABLE LINKING

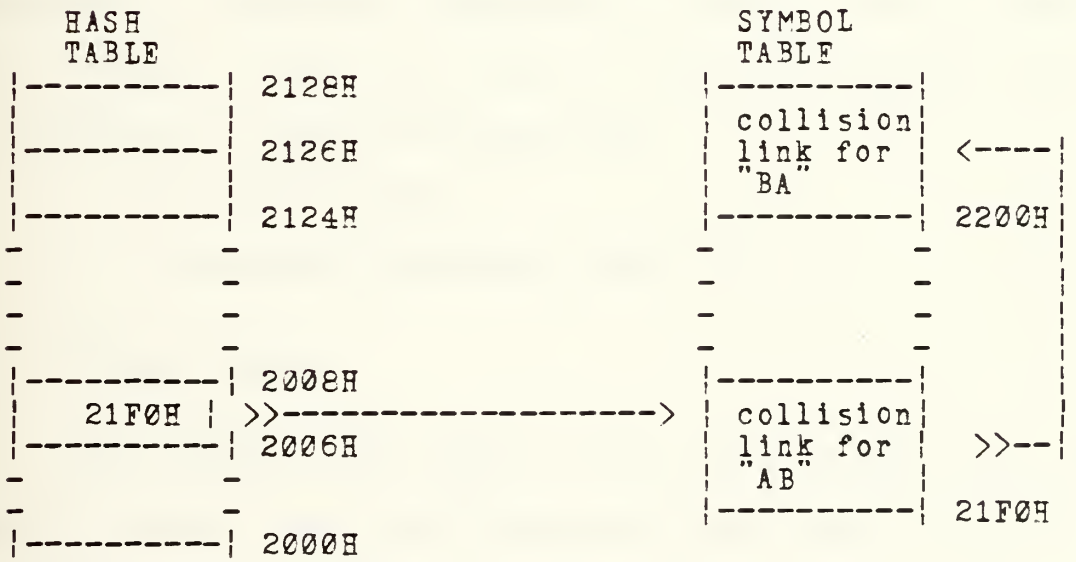


FIGURE II-2

1. Numeric Values

The symbol table entry for numeric values can contain up to eight attributes of the variable. These attributes are: 1.) identifier type, 2.) length of variable name 3.) beginning address of variable storage, 4.) numeric count (number of storage locations required by the identifier), 5.) level number, 6.) number of digits to the right of the decimal point, 7.) the variable name, and 8.) a previous occurs pointer. The previous occurs pointer is appended after the identifier name only if needed. Since most declarations will not require the use of this pointer, a saving of three bytes per variable declaration is realized. It was felt that the increase in the total number of variables that could be declared in a given memory size outweighed the increased complexity in symbol table access time. Figures [II-3] and [II-4] illustrate, respectively, the following two COBOL declarations:

```
Ø1 NUM PIC 9(9).
Ø2 NUM PIC 9(6)V999 OCCURS 12.
```

2. Numeric Edit

The numeric edit symbol table entry expands on the numeric symbol entry and utilizes bytes eight and nine to hold the beginning address, in the constants area, of the edit field mask. This mask allowed for the insertion of the

following characters into the output display of a numeric number: fixed and floating dollar signs, credit(CR) and debit(DB) signs, asterisk fill, "Z" character fill, and plus ("+") and minus ("-") signs. It should be noted that an identifier with a numeric edit field value can not be used in an arithmetic statement. Figure [II-5] illustrates the following COBOL declaration:

```
Ø1 NUM PIC +$ZZZ,ZZ9.99.
```

NUMERIC SYMBOL TABLE ENTRY.

BYTE	SYMBOL TABLE VALUE
Ø-1	collision link (ØØ ØØ)
2	type identifier (1Ø)
3	length of identifier name (Ø3)
4-5	beginning address of identifier storage (Ø4 25)
6-7	length of identifier storage (Ø9 ØØ)
8-9	not used
1Ø	level entry (Ø1)
11	decimal count (ØØ)
12-13	occurances (ØØ)
14-16	identifier name (4E 55 4D)

Ø1 NUM PIC 9(9).

FIGURE II-3

NUMERIC SYMBOL TABLE ENTRY WITH DECIMAL AND OCCURS CLAUSE

BYTE	SYMBOL TABLE VALUE
0-1	collision link (09 2E)
2	type identifier (10)
3	length of identifier name (03)
4-5	beginning address of identifier storage (0D 25)
6-7	length of identifier storage (09 00)
8-9	not used
10	level entry (02)
11	decimal count (03)
12-13	occurances (0C)
14-16	identifier name (4E 55 4D)
17-18	previous occurs pointer 00 00
19	dimension counter

02 NUM PIC 9(6)V999 OCCURS 12.

FIGURE II-4

NUMERIC SYMBOL TABLE ENTRY WITH EDITED FIELD

BYTE	SYMBOL TABLE VALUE
0-1	colision link (09 2E)
2	type identifier (80)
3	length of identifier name (03)
4-5	beginning address of identifier storage (0D 25)
6-7	length of identifier storage (09 00)
8-9	beginning address of mask storage (25 FE)
10	level entry (01)
11	decimal count (02)
12-13	occurances (00)
14-16	identifier name (4E 55 4D)

01 NUM PIC +$ZZZ,ZZ9.99.

FIGURE II-5

3. Alpha or Alphanumeric

The alpha and alphanumeric symbol table entries appear similarly in the symbol table except for their type fields. Six entries appear in the symbol table for these identifiers: 1.) identifier type, 2.) length of identifier name, 3.) beginning address of storage, 4.) number of storage locations required by the identifier, 5.) level entry, and 6.) identifier name. Figure [II-6] illustrates an alpha symbol table entry for the following identifier declaration:

Ø1 ALPHA PIC A(8).

4. Alpha Edit

The alpha edit symbol table entry expands on the alpha and alphanumeric edit types and utilizes bytes eight and nine to hold the beginning address of the edit field mask. These mask fields, which are stored in the constants area of the pseudo-machine, contain the characters necessary to edit an output so that, for example, slashes or blanks can be interspersed in the display output.

ALPHA SYMBOL TABLE ENTRY

BYTE	SYMBOL TABLE VALUE
0-1	collision link (00 00)
2	type identifier (08)
3	length of identifier (05)
4-5	beginning address of identifier storage (16 25)
6-7	length of identifier storage (08 00)
8-9	not used
10	level entry (01)
11	not used
12-13	not used
13-17	identifier name (41 4C 50 48 41)

01 ALPHA PIC A(8).

FIGURE II-6

5. Tables

NPS MICRO-COBOL supports multiply indexed tables up to a maximum of ten levels. The choice of ten levels was based on a compromise between a single level of HYPO-COBOL and 49 levels proposed for the new 1980 ANSI COBOL standard. The limit of ten levels is a restriction for HYPO-COBOL and the nucleus level 1 constructs of ANSI-COBOL. These tables are established by using an OCCURS clause with the PIC clause of an identifier. If an identifier is specified as a table the number of occurances of the table are placed in byte twelve and thirteen of the symbol table entry for that identifier. The table identifier in COBOL is similar to the subscripted variable in other programming languages. The previous occurs pointer shown in FIGURE II-4 is used to indicate where variables are located and how many occurances exist to enable the compiler to generate the proper base address. For example, the statement, "02 NUM PIC 9(9) OCCURS 12", generates the symbol table entry illustrated in figure [II-4].

6. Labels

Labels generate the simplest of all symbol table entries, only four or five attributes are associated with the label. The variability depends on whether the label is declared in the source program before or after the label is

referenced by a GO or PERFORM statement. In the event that a label is specified before a GO or PERFORM statement references it, the symbol table would contain the following 1.) the type associated with label, 2.) the length of the identifier name, 3.) the address of the first intermediate code instruction following the appearance of the label in the source program (bytes four and five), 4.) the last executable instruction associated with the label (bytes eight and nine) (This would be either the last executable instruction encountered before another label or the end of the program), and 5.) the label name.

In the event a label is referenced by a GO or PERFORM statement before the label actually appears in the code, the symbol table entry performs a different function than just indicating the beginning and ending of the paragraph associated with the label. The same symbol table fields are used, however their meanings are different. The type is set to that of an unresolved label(ØFFH). The label remains unresolved until the beginning and the ending addresses of the associated paragraph are determined. If a label is never resolved by the end the input file, an error for each unresolved label is produced as a warning to the user.

When a label is referenced for the first time by a GO statement the symbol table is initialized with the following: 1.) unresolved label type (ØFFH), 2.) the address

of the GO statement (the intermediate code would be BRN ØØ ØØ where the zeros indicate where the address of the label is to be backstuffed. See section III-D for specific explanation of pseudo-machine instructions), 3.) the remainder of the label entries would be the same except no entry is made for the last executable instruction associated with the label. If an additional reference is made to the label by a subsequent GO statement the following action would occur: 1.) the current address (bytes four and five) would be placed in the branch address of the GO statement, 2.) the address of this branch statement would be placed in bytes four and five of the symbol table entry. This procedure facilitates linking together all unresolved references to labels so as a result when the label is resolved the correct branch address can easily be placed into the intermediate code.

Encountering a PERFORM statement before a label is declared causes the following actions: 1.) Bytes four and five contain the address of the next byte of intermediate code following the PER intermediate code instruction, and 2.) bytes eight and nine contain the address of the third byte following the PER instruction. If a subsequent PERFORM statement is encountered before the label is resolved the two address fields in the symbol table would be copied to the associated bytes following the most current PERFORM statement and the address of the first and third bytes

following the PER instruction would be copied into the symbol table. It should be pointed out that any number of PERFORM and GO statements can be specified before a label is resolved.

7. Files

The symbol table entries for files are the most difficult to understand. The complexity of the entries is due to the way files and records are declared in a MICRO-COBOL program. The symbol table entry for a file consists of the following: 1.) byte two contains the type, 2.) byte three contains the length of the file name, 3.) bytes four and five contain the address in the symbol table of the first Ø1 level record associated with the file, 4.) bytes eight and nine contain the beginning address of the file control block and input/output buffer for the file, (this would be the actual address in the data section of the pseudo-machine for the beginning of the 165 bytes associated with the file), 5.) if the file has a key entry associated with it (access via RANDOM or RANDOM RELATIVE) bytes ten and eleven contain the symbol table address of the access key variable, and 6.) the rest of the entry contains the file name. Figure [II-6] illustrates a file entry in the symbol table.

8. Records

This entry contains seven attributes of a record. Three are the same as all other entries type, name, and length of name. While the other four are: 1.) bytes four and five contain the initial storage address for the record, 2.) bytes six and seven contain the number of bytes of storage for the record, 3.) bytes eight and nine contain the symbol table address of the file associated with the record (this facilitates referencing the file when the record is written), and 4.) byte ten contains the level entry for the record.

FILE SYMBOL TABLE ENTRY

SAMPLE SOURCE PROGRAM FILE DECLARATION

```
INPUT-OUTPUT SECTION.
FILE-CONTROL.
      SELECT ROSTER-FIL
             ORGANIZATION RELATIVE
             ACCESS RANDOM RELATIVE NUM
             ASSIGN CS81-FIL.
```

BYTE	SYMBOL TABLE VALUE
Ø-1	collison link
2	type file (Ø3)
3	length of file name (Ø5)
4-5	symbol table address of first Ø1 level record (Ø9 2E)
6-7	not used
8-9	first address of FCB & buffer (ØE 26)
1Ø-11	symbol table address of key (33 27)
12-13	not used
14-18	file name (52 4F 53 54 45 52 5F 46 49 4C)

FIGURE II-7

C. COMPILER MODULE "PART ONE"

1. Purpose

The first module of the compiler performs several functions. First, it establishes the interface between the compiler and: 1.) the input source file (of type "CBL"), 2.) the output intermediate code file (of type "CIN"), 3.) the output list file (of type "LST"), and 4.) the READER module which reads and passes control to PART TWO of the compiler. Second, it scans and parses the source program statements up to the PROCEDURE DIVISION. Third, it generates output consisting of the symbol table entries (saved in memory) and data initialization intermediate code. A listing file is also created which will contain any compilation errors generated and a listing of the source code if the appropriate toggle is activated. See Appendix A for a list of compiler options.

2. Control Actions

By executing the command COBOL <source program> $<compiler toggles> the object code for PART ONE of the compiler is loaded into memory starting at 100H (if necessary this can be modified for different machines) by the CP/M operating system. Execution of PART ONE loads the source program name into the input file control block located at 5CH. This allows the source program name to be

saved until actual source program compilation begins. The compiler toggles are loaded into the input file control block located at 6CH. These optional toggles are used later to initialize certain features such as code, nocode, list, nolist, etc. See Appendix A for a complete list of options.

Next, the control program, READER, is moved to high memory just below the BDOS (see reference 4 for an explanation of BDOS and other CP/M associated names). For example, using an INTEL Corporation 62K MDS microcomputer system with the CP/M operating system, the READER routine is moved to high memory starting at ØDØØØH and continuing through ØDØFFH. This is done for two reasons: 1.) it allows the symbol table of the source program to begin at the next address following the object code for PART ONE, and 2.) places READER high enough in memory so that it is not destroyed by creation of the symbol table. See figures [II-7] and [II-8] for illustrations of the PART ONE memory organization before and after the READER routine is moved. The purpose of the READER routine will be explained in the next section.

MEMORY ORGANIZATION BEFORE READER ROUTINE MOVED

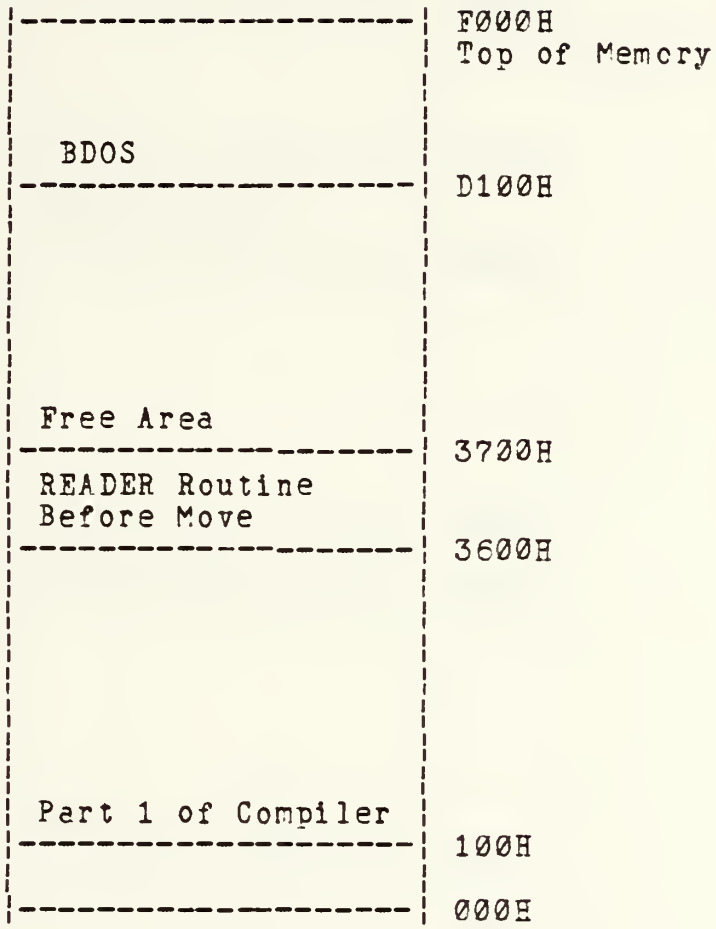


FIGURE II-7

MEMORY ORGANIZATION AFTER READER ROUTINE MOVED

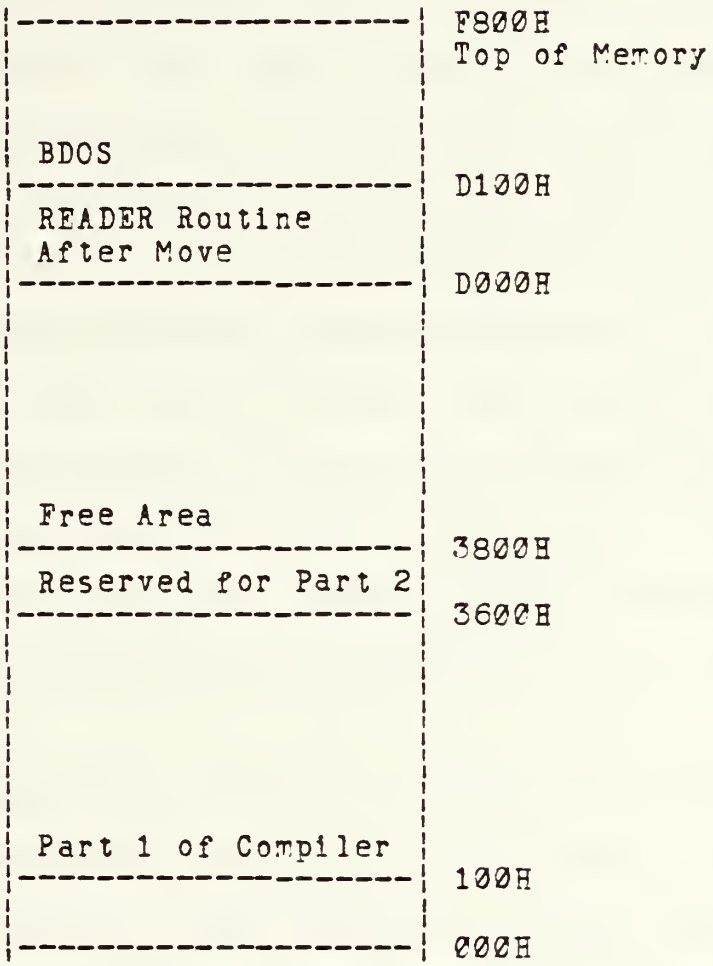


FIGURE II-8

Next, the interface between the compiler and the input file <source program> and the output file <intermediate code file> is established. The input file control block associated with the source file is initialized and the input file is opened. The input file name is copied to the output file control block (FCB) and if there is an intermediate code file already residing on the disk, it is erased. The output FCB is initialized and a file directory entry established for the new copy of the intermediate code file. A list file control block and associated buffer are created and opened. The list file contains any error messages generated by the compiler and the line being parsed at the time the error was discovered. The relative line number is also provided. With the list toggle activated the list file will contain the complete input file with errors and line numbers.

Prior to beginning scanning and parsing actions, the first 128 byte record of the input file is read into the input buffer, located at 80H (default I/O buffer for CP/M). The scanner is primed with the first character of the input program, and scanning and parsing actions continue from this point in PART ONE until the PROCEDURE DIVISION of the source program is encountered; at this time compilation is suspended.

3. Symbol Table Entries

Entries made in the symbol table by PART ONE will consist of all identifiers declared in the DATA DIVISION of the source program. By refering to the Symbol Table Section above, an explanation may be obtained regarding the various types of symbol table entries.

4. Intermediate Code Generation

Pseudo-instructions are written to the intermediate code file for several different reasons while PART ONE is scanning and parsing the source program. The first intermediate code generated occurs when the INPUT-OUTPUT SECTION of a source program is nonempty. Within the FILE CONTROL PARAGRAPH of this section, instructions are generated to initialize the FCB for the file name associated with the SELECT statement. The name associated with the ASSIGN statement is placed in the FCB and is used in referencing the file on the disk.

Two other instances of intermediate code generation occur in the WORKING STORAGE SECTION of a source program. Anytime a record or elementary identifier entry has an edited PICTURE CLAUSE, code to initialize the storage beginning at the address specified in the formatted mask attribute of the symbol table entry will be written to the intermediate code file. When a record or elementary identifier entry has an associated numeric or nonnumeric

VALUE CLAUSE, code to initialize the storage beginning at the address specified in the value location attribute of the symbol table entry will be written to the intermediate code file.

The final pseudo-instruction written to the intermediate code file is the SCD instruction. This occurs when the parser parses the word PROCEDURE in the source program; control is then passed to PART TWO and compilation continues.

5. Parser Actions

The actions corresponding to each parse step are explained below. In each case, the grammar rule that is being applied is given, and an explanation of what program actions take place for that step has been included. In describing the actions taken for each parse step there has been no attempt to describe how the symbol table is constructed, what pseudo-instructions are generated or how the values are preserved on the stack. The intent of this section is to describe what information needs to be retained and at what point in the parse it can be determined. Where no action is required for a given statement, or where the only action is to save the contents of the top of the stack, no explanation is given. Questions regarding the actual manipulation of information should be resolved by consulting the program listings.

```
1  <program> ::= <id-div> <e-div> <d-div> PROCEDURE
     Reading the word PROCEDURE terminates the first
     part of the compiler.
2  <id-div> ::= IDENTIFICATION DIVISION. PROGRAM-ID.
                 <comment> . <id-list>
3  <id-list> ::= <auth> <ins> <date> <sec>
4  <auth> ::= AUTHOR . <comment> .
5           | <empty>
6  <ins> ::= INSTALLATION . <comment> .
7            | <empty>
8  <date> ::= DATE-WRITTEN . <comment> .
9           | <empty>
10  <sec> ::= SECURITY . <comment> .
11          | <empty>
12  <comment> ::= <input>
13              | <comment> <input>
14  <e-div> ::= ENVIRONMENT DIVISION . CONFIGURATION
                SECTION. <src-obj> <i-o>
15              | <empty>
16  <src-obj> ::= SOURCE-COMPUTER . <comment> <debug> .
                  OBJECT-COMPUTER . <comment> .
17  <debug> ::= DEBUGGING MODE
      Set a scanner toggle so that debug lines will be
      read.
18            | <empty>
19  <i-o> ::= INPUT-OUTPUT SECTION . FILE-CONTROL .
```

```
            <file-control-list> <ic>
20        ¦ <empty>
21  <file-control-list> ::= <file-control-entry>
22                        ¦ <file-control-list>
                              <file-control-entry>
23  <file-control-entry> ::= SELECT <id> <attribute-list> .
```

At this point all of the information about the file has been collected and the type of the file can be determined. File attributes are checked for compatibility and entered in the symbol table.

```
24  <attribute-list> ::= <one attrib>
25                     ¦ <attribute-list> <one attrib>
26  <one-attrib> ::= ORGANIZATION <org-type>
27                 ¦ ACCESS <acc-type> <relative>
28                 ¦ ASSIGN <input>
```

A file control block is built for the file using the INT operator.

```
29  <org-type> ::= SEQUENTIAL
```

No information needs to be stored since the default file organization is sequential.

```
30                ¦ RELATIVE
```

The relative attribute is saved for production 23.

```
31                ¦ INDEXED
```

The indexed attribute is not implemented.

```
27  <acc-type> ::= SEQUENTIAL
```

This is the default.

28 | RANDOM

The random access mode is saved for production 19.

29 <relative> ::= RELATIVE <id>

The pointer to the identifier will be retained by the current symbol pointer, so this production only saves a flag on the value stack indicating that the production did occur.

35 | <empty>

36 <ic> ::= I-O-CONTROL . <same-list>

37 | <empty>

38 <same-list> ::= <same-element>

39 | <same-list> <same-element>

40 <same-element> ::= SAME <id-string> .

41 <id-string> ::= <id>

42 | <id-string> <id>

43 <d-div> ::= DATA DIVISION . <file-section> <work> <link>

44 <file-section> ::= FILE SECTION . <file-list>

A flag needs to be set to indicate completion of the file section, so that the appropriate routine will be called when parsing level entries in the WORKING STORAGE SECTION.

45 | <empty>

The flag, indicated in production 44, is set.

46 <file-list> ::= <file-element>

47 | <file-list> <file-element>

48 <files> ::= FD <id> <file-control> .
<record-description>

This statement indicates the end of a record description, if there was an implied redefinition of the record, then the level stack (ID$STACK) must be reduced. The length of the first record description and its address can now be loaded into the symbol table for the file name.

49 <file-control> ::= <file-list>

The address of the symbol table entry for the record describing the file name is entered into an attribute of the file name symbol table entry, while the address of the file name's symbol table entry is entered into an attribute of the same record.

50 | <empty>

Same as 49 above.

51 <file-list> ::= <file-element>

52 | <file-list> <file-element>

53 <file-element> ::= BLOCK <integer> RECORDS

54 | RECORD <rec-count>

The record length is saved for comparison with the calculated length from the picture clauses.

55 | LABEL RECORDS STANDARD

56 | LABEL RECORDS OMITTED

57 | VALUE OF <id-string>

58 <rec-count> ::= <integer>

59 | <integer> TO <integer>

The TO option is the only indication that the file will be variable length. The maximum length must be saved.

60 <work> ::= WORKING-STORAGE SECTION . <record-description>

If the level stack (ID$STACK) contains a record identifier with a level number greater than one, then the stack must be reduced. The reduction depends on whether the identifier on the top of the stack is a redefinition of the item beneath it or not. The primary action is to assign the proper amount of storage to the last record in the WORKING STORAGE SECTION.

61 | <empty>

62 <link> ::= LINKAGE SECTION . <record-description>

63 | <empty>

64 <record-description> ::= <level-entry>

65 |<record-description> <level-entry>

66 <level-entry> ::= <integer> <data-id> <redefines> <data-type> .

The symbol table address for the level entry identifier is loaded into the level stack (ID$STACK). The level stack keeps track of the nesting of field definitions (elementary items) in a record in the FILE and WORKING STORAGE

SECTIONS. At this point there may be no information about the length of the item being defined and its attributes may depend entirely upon its constituent fields. Within the FILE SECTION, multiple record descriptions for a file are assumed to be redefinitions of the first record description. In the WORKING STORAGE SECTION, if there is a VALUE CLAUSE, the stack level to which it applies is saved in PENDING$LITERAL, the level entry number is saved in VALUE$LEVEL and a flag, VALUE$FLAG, is set.

67 <data-id> ::= <id>

68 | FILLER

An entry is built in the symbol table to record information about this record field. It cannot be used explicitly in a program because it has no name, but its attributes will need to be stored as part of the total record.

69 <redefines> ::= REDEFINES <id>

The redefines option gives new attributes to a previously defined record area. The symbol table pointer to the area being redefined is saved in an attribute of the redefining identifier's symbol table entry, so that information can be transferred to the area by either identifier. In addition to the information saved relative to the redefinition, it is nec-

essary to check to see if the current identifier's level number is less than or equal to the level number of the identifier currently on the top of the level stack. If this is true, then all information for the item on top of the stack has been saved and the stack can be reduced. If the current identifier is a redefinition of another identifier, the stack entry for the record being redefined is not removed until the first non-redefinition of a current identifier at the same level.

70 | <empty>

As in production 64, the stack (ID$STACK) is checked to determine if the current level number indicates a reduction of the level stack is necessary. In addition, special action needs to be taken if the new level is 01. If an 01 level is encountered at this production prior to production 39 or 40 (the end of the file area), it is an implied redefinition of the previous 01 level record. In the WORKING STORAGE SECTION, it indicates the start of a new record.

71 <data-type> ::= <prop-list>

72 | <empty>

73 <prop-list> ::= <data-element>

74 | <prop-list> <data-element>

75 <data-element> ::= PIC <input>

The <input> at this point is the character string

that defines the record field. It is analyzed and the necessary extracted information is stored in the symbol table.

76 | USAGE COMP

The field is defined as a binary field; however, COMP has not been implemented, therefore, if there is an associated VALUE CLAUSE, the value is entered into the associated identifier's value storage location in display format.

77 | USAGE COMP-3

The field is defined as a packed Binary Coded Decimal field.

78 | USAGE COMPUTATIONAL

Optional form of USAGE COMP.

79 | USAGE DISPLAY

The DISPLAY format is the default, and thus no special action occurs.

80 | SIGN LEADING <separate>

This production indicates the presence of a sign in a numeric field. The sign will be in a leading position. If the <separate> indicator is true, then the length will be one longer than the PICTURE CLAUSE, and the type will be changed to signed numeric leading and separate.

81 | SIGN TRAILING <separate>

The same information required by production 73 must

be recorded, but in this case the sign is trailing rather than leading.

82 | OCCURS <integer> INDEXED <id>

83 | OCCURS <integer>

The type must be set to indicate multiple occurrences and the number of occurrences saved for computing the space defined by this field.

84 | SYNC <direction>

Syncronization with a natural boundary is not required by this machine.

85 | VALUE <literal>

The field being defined will be assigned an initial value determined by the value of the literal through the use of an INT operator. This is only valid in the WORKING-STORAGE SECTION. Note that numeric and signed numeric PICTURE CLAUSES will have a numeric -- no quotes delimiting -- VALUE CLAUSE, while alphanumeric and alpha types will have a nonnumeric -- literal delimited with quotes -- VALUE CLAUSE.

86 <direction> ::= LEFT

87 | RIGHT

88 | <empty>

89 <separate> ::= SEPARATE

The separate sign indicator is set.

90 | <empty>

91 <literal> ::= <input>

The input string is checked to see if it is a valid numeric literal, and if valid, it is stored to be used in a value assignment.

92 | <lit>

This literal is a quoted string.

93 | ZERO

As the case of all literals, the fact that there is a pending literal needs to be saved. In this case and the three following cases, an indicator of which literal constant is being saved is all that is required. The literal value can be reconstructed later.

94 | SPACE

95 | QUOTE

96 <integer> ::= <input>

The input string is converted to an integer value for later internal use.

97 <id> ::= <input>

The input string is the name of an identifier and is checked aginst the symbol table. If it is in the symbol table, then a pointer to the entry is saved. If it is not in the symbol table, then it is entered and the address of the entry is saved.

D. INTERFACE ACTIONS

When compilation is suspended in PART ONE of the compiler certain key variables are saved for use in PART TWO. These variables are declared sequentially in PART ONE and are therefore located in contiguous memory in the variable area of PART ONE. These variables consist of debugging toggles set when invoking the compiler, i.e. sequence or token numbers, a pointer to the next available address in the symbol table, a pointer to the next character in the input source file, the output and list file control blocks, the output and list buffers, the error counter, the next address in the intermediate code area, the next address in the constants area, and the base address of the symbol table. These key variables, consisting of 353 bytes, are copied to the 353 bytes immediately below the READER routine to insure they are not destroyed when PART TWO of the compiler is brought into memory. Since the memory area required for PART ONE is larger than that required by PART TWO the symbol table does not need to be relocated. Since the symbol table is not altered when PART TWO of the compiler is brought into memory only the base address of the symbol table and the last address of the symbol table need be saved to insure that access to the symbol table can be continued in PART TWO. See Figure [II-10] for an illustration of the memory organization when control is transfered from PART ONE to READER. The READER routine causes PART TWO of the compiler to be brought into memory

starting at 100H and then transfers control to PART TWO of the Compiler.

E. COMPILER MODULE "PART TWO"

1. Purpose

The second part of the compiler scans and parses the MICRO-COBOL source statements starting with the PROCEDURE DIVISION and generates the necessary intermediate code.

2. Control Actions

The first action after control is transfered to PART TWO from the READER routine is to copy the 353 bytes of information saved from PART ONE into associated variables in PART TWO. After these variables are initialized all references to files, symbol table entries, etc. can be made in PART TWO and compilation can continue. See Figure [II-11] for an illustration of the memory organization at the time PART TWO begins compilation.

3. Symbol Table Entries

Entries made in the symbol table by PART TWO will be those for paragraph labels encountered within the PROCEDURE DIVISION of the source program.

4. Intermediate Code Generation

For an explanation of the pseudo-instructions that are generated by PART TWO refer to the compiler program listings and the parser actions below. Also, for general information on pseudo-instructions refer to section III-D.

MEMORY ORGANIZATION WHEN CONTROL IS TRANSFERED TO READER

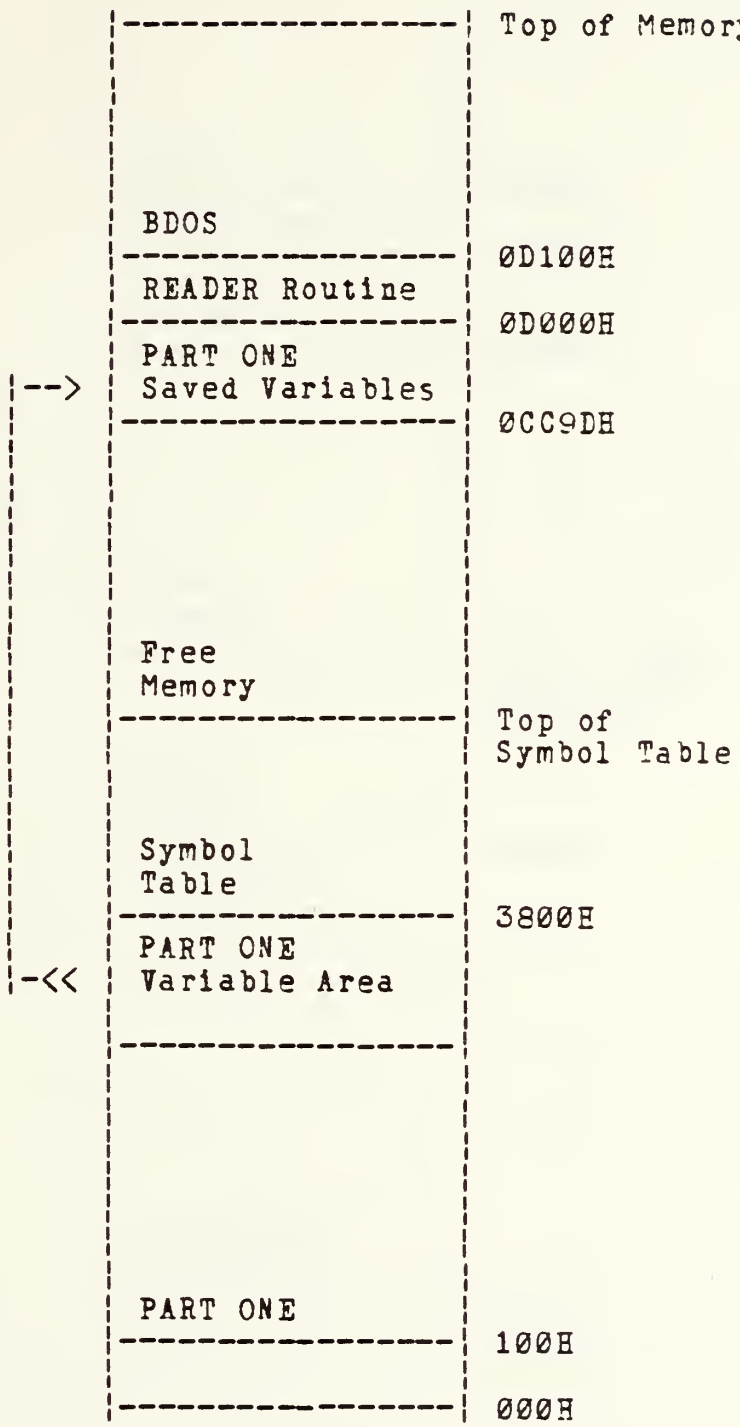


FIGURE II-10

MEMORY ORGANIZATION AFTER PART TWO IS COPIED INTO MEMORY

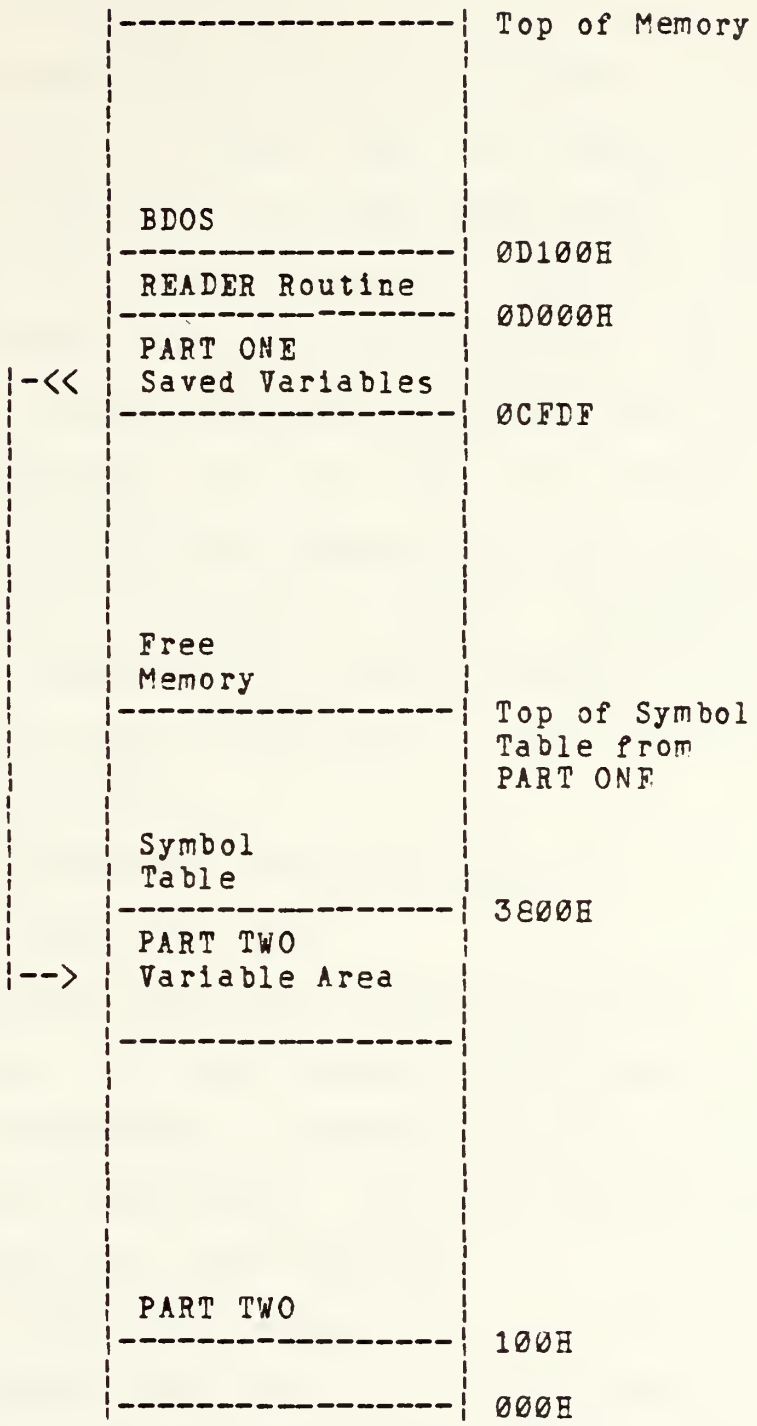


FIGURE II-11

5. Parser Actions

The actions corresponding to each parse step in PART TWO are explained below. In each case, the grammar action that is being applied is given, and an explanation of what program actions take place for that step has been included. In describing the actions taken for each parse step there has been no attempt to describe how the symbol table entries are made, what pseudo instructions are generated or how the values are preserved on the stack. The intent of this section is to describe what information needs to be retained and at what point in the parse it can be determined. Where no action is required for a given statement, or where the only action is to save the contents of the top of the stack, no explanation is given.

1 <p-div> ::= PROCEDURE DIVISION <using> .
<proc-body> EOF

This production indicates termination of the compilation. If the program has sections, then it will be necessary to terminate the last section with a RET Ø instruction. The code will be ended by the output of a TER operation.

2 <using> ::= USING <id-string>

If the reserved word CALL is on the procedure stack then the PAR operator is produced followed by the addresses of the parameters that will be passed from the calling

program. If the reserved words PROCECEDURE DIVISION are on the procedure stack then the identifier stack contains the formal parameters that will be used for that procedure. These variables are given sequential address locations starting at ØDH so that the addresses may be resolved at run time by getting the actual parameter address off the call stack.

PAR <number of parameters> <parameter #1 address> ...

3 ¦ <empty>

4 <id-string> ::= <id>

The identifier stack is cleared and the symbol table address of the identifier is loaded into the first stack location.

5 ¦ <id-string> <id>

The identifier stack is incremented and the symbol table pointer stacked.

6 <proc-body> ::= <paragraph>

7 ¦ <proc-body> <paragraph>

8 <paragraph> ::= <id> .

9 ¦ <id> . <sentence-list>

The starting and ending address of the paragraph are entered into the symbol table. A return is emitted as the last instruction in the paragraph (RET Ø). When the label is resolved, it may be necessary to produce a BST operation to resolve previous references to the label.

10 ¦ <id> SECTION .

The starting address for the section is saved. If it is not the first, then the previous section ending address is loaded and a return (RET Ø) is output. As in production 9, a BST may be produced.

11 <sentence-list> ::= <sentence> .

12 ¦ <sentence-list> <sentence> .

13 <sentence> ::= <imperative>

14 ¦ <conditional>

15 ¦ ENTER <id> <opt-id>

This construct is not implemented. An ENTER allows statements from another language to inserted in the source code.

16 <imperative> ::= ACCEPT <subid>

ACC <address> <length>

17 ¦ <arithmetic>

18 ¦ CALL <call-lit> <using>

The SBR operator is produced.

SBR <subroutine name>

19 ¦ CLOSE <close-lst>

CLS <file control block address>

20 ¦ <file-act>

21 ¦ DISPLAY <display-lst>

The display operator is produced for the first literal or identifier.

DIS <address> <length> <flag>

22 | DISPLAY <display-lst> WITH NO ADVANCING

The DISPLAY WITH NO ADVANCING option is not implemented.

23 | EXIT <program-id>

RET Ø

24 | GO <id>

BRN <address>

25 | GO <id-string> DEPENDING <id>

GDP is output, followed by a number of parameters: <the number of entries in the identifier stack> <the length of the depending identifier> <the address of the depending identifier> <the address of each identifier in the stack>.

26 | MOVE <lit/id> TO <subid>

The types of the two fields determine the move that is generated. Numeric moves go through register two using a load and a store. Non-numeric moves depend upon the resultant field and may be either MOV, MED or MNE. Since all of these instructions have long parameter lists, they have not been listed in detail.

27 | OPEN <act-lst>

28 | PERFORM <id> <thru> <finish>

The PER operation is generated followed by the <branch address> <the address of the return

statement to be set> and <the next instruction address>.

29 ¦ STOP <terminate>

If there is a terminate message, then STD is produced followed by <message address> <message length>. Otherwise STP is emitted.

30 <close-lst> ::= <id>

31 ¦ <close-lst> <id>

Multiple close option is not implemented.

32 <display-lst> ::= <lit/id>

33 ¦ <display-lst> <lit/id>

Multiple display option is not implemented.

34 <act-lst> ::= <type-action> <open-lst>

This produces either OPN, OP1, or OP2 depending upon the <type-action>. Each of these is followed by file control block address.

35 ¦ <act-lst> <type-action> <open-lst>

36 <open-lst> ::= <id>

37 ¦ <open-lst> <id>

Multiple open option is not implemented.

38 <finish> ::= <1/id> TIMES

This produces the code to perform a paragraph <1/id> TIMES.

39 ¦ <stopcondition>

40 ¦ <varying> <iteration> <stopcondition>

41 ¦ <empty>

42 <stopcondition> ::= UNTIL <condition>

43 <varying> ::= VARYING <subid>

44 <iteration> ::= <from> <by>

45 <from> ::= FROM <l/id>

The counter is initialized to <l/id>.

46 <by> ::= BY <l/id>

The counter is incremented BY <l/id>.

47 <conditional> ::= <arithmetic> <size-error> <imperative>

A BST operator is output to complete the branch around the imperative from production 117.

48 | <file-act> <invalid> <imperative>

A BST operator is output to complete the branch from production 116.

49 | <read-id> <special> <imperative>

A BST is produced to complete the branch around the <imperative>.

50 | <if-nonterminal> <condition>
<if-lst> <else> <if-lst> END-IF

NEG will be emitted unless <condition> is a "NOT <cond-type>", in which case the two negatives will cancel each other. Two BST operators are required. The first fills in the branch to the ELSE action. The second completes the branch around the <if-lst> which follows ELSE.

51 | <if-nonterminal> <condition>
<if-lst> END-IF

52 <if-lst> ::= <stmt-lst>

53 ¦ NEXT SENTENCE

A branch operator is produced to branch to the end of the current sentence.

54 <else> ::= ELSE

55 <Arithmetic> ::= ADD <add-lst> TO <subid> <round>

The existence of multiple load and store instructions make it difficult to indicate exactly what code will be generated for any of the arithmetic instructions. The type of load and store will depend on the nature of the number involved, and in each case the standard parameters will be produced. This parse step will involve the following actions: first, a load will be emitted for the first number into register zero. If there is a second number, then a load into register one will be produced for it, followed by an ADD and a STI. Next a load into register one will be generated for the result number. Then an ADD instruction will be emitted. Finally, if the round indicator is set, a RND operator will be produced prior to the store.

56 ¦ ADD <add-lst> GIVING <subid> <round>

The ADD GIVING option is not implemented.

57 ¦ DIVIDE <1/id> INTO <1/id> <round>

The first number is loaded into register zero. The second operand is loaded into register one. A DIV operator is generated, followed by a RND operator prior to the store, if required.

```
58                  | DIVIDE <l/id> BY <l/id> GIVING
                      <subid> <round>
```

The DIVIDE GIVING option is not implemented.

```
59                  | DIVIDE <l/id> INTO <l/id> GIVING
                      <subid> <round>
60                  | MULTIPLY <l/id> BY <subid> <round>
```

The multiply is the same as the divide except that a MUL operator is generated.

```
61                  | MULTIPLY <l/id> BY <l/id> GIVING
                      <subid> <round>
62                  | SUBTRACT <sub-lst> FROM <subid>
                      <round>
```

Subtaction generates the same code as the ADD except that a SUB is produced in place of the ADD.

```
63                  | SUBTRACT <sub-lst> GIVING <subid>
                      <round>
```

The SUBTRACT GIVING option is not implemented.

```
64                  | COMPUTE <subid> = <arith-exp>
```

The COMPUTE verb is not implemented.

```
65  <add-lst> ::= <l/id>
66                | <add-lst> <l/id>
```

Multiple ADD option is not implemented.

```
67  <sub-lst> ::= <l/id>
68                | <sub-lst> <l/id>
```

Multiple SUBTRACT option is not implemented.

```
69  <arith-exp> ::= <term>
```

Productions 69 through 80 are required for the COMPUTE verb and are not implemented.

```
70                  | <arith-exp> + <term>
71                  | <arith-exp> - <term>
72                  | + <term>
73                  | - <term>
74  <term> ::= <primary>
75           | <term> * <primary>
76           | <term> / <primary>
77  <primary> ::= <prim-elem>
78              | <primary> ** <prim-elem>
79  <prim-elem> ::= <l/id>
80                  | ( <arith-exp> )
81  <file-act> ::= DELETE <id>
```

Either a DLS or a DLR will be produced along with the required parameters.

```
82                  | REWRITE <id>
```

Either a RWS or a RWR is emitted, followed by parameters.

```
83                  | WRITE <id> <special-act>
```

There are four possible write instructions: WTF, WVL, WRS, and WRR.

```
84  <condition> ::= <bterm>
```

The logical OR and AND operators are not implemented.

```
85                  | <condition> OR <bterm>
86  <bterm> ::= <bprim>
```

87 ¦ <bterm> AND <bprim>

88 <bprim> ::= <lit/id>

89 ¦ <lit> <not> <cond-type>

One of the compare instructions is produced. They are CAL, CNS, CNU, RGT, RLT, REQ, SGT, SLT, and SEQ. Two load instructions and a SUB will also be generated if one of the register comparisons is required.

90 ¦ (<bterm>)

91 <cond-type> ::= NUMERIC

92 ¦ ALPHABETIC

93 ¦ <compare> <lit/id>

94 <not> ::= NOT

NEG is emitted unless the NOT is part of an IF statement in which case the NEG in the IF statement is cancelled.

95 ¦ <empty>

96 <compare> ::= GREATER

97 ¦ LESS

98 ¦ EQUAL

99 ¦ >

Productions 99-101 are not implemented.

100 ¦ <

101 ¦ =

102 <ROUND> ::= ROUNDED

103 ¦ <empty>

104 <terminate> ::= <literal>

```
105                 ¦ RUN
106  <special> ::= <invalid>
107                ¦ END
```

An EOR operator is emitted followed by a zero. The zero acts as a filler in the code and will be back-stuffed with a branch address. In this production and several of the following, there is a forward branch on a false condition past an imperative action. For an example of the resolution, examine production 48.

```
108  <opt-id> ::= <subid>
109               ¦ <empty>
110 <stmt-lst> ::= <imperative>
111                ¦ <stmt-lst> <imperative>
112                ¦ <conditional>
113                ¦ <stmt-lst> <conditional>
114  <thru> ::= THRU <id>
115            ¦ <empty>
116  <invalid> ::= INVALID
      INV Ø
117  <size-error> ::= SIZE ERROR
      SER Ø
118  <special-act> ::= <when> ADVANCING <how-many>
119                    ¦ <empty>
120  <when> ::= BEFORE
121            ¦ AFTER
122  <how-many>::= <integer>
```

123 ¦ PAGE

124 <type-action> ::= INPUT

125 ¦ OUTPUT

126 ¦ I-O

127 <subid> ::= <subscript>

128 ¦ <id>

129 <integer> ::= <input>

The value of the input string is saved as an internal number.

130 <id> ::= <input>

The identifier is checked against the symbol table, if it is not present, it is entered as an unresolved label.

131 <l/id> ::= <input>

The input value may be a numeric literal. If so, it is placed in the constant area with an INT operator. If it is not a numeric literal, then it must be an identifier, and it is located in the symbol table.

132 ¦ <subscript>

133 ¦ ZERO

134 <subscript> ::= <id> (<subscript-lst>)

A SCR operator is produced with the base address of a variable defined with an OCCURS clause. Multiple subscripting has not been implemented.

135 <subscript-lst> ::= <input>

136 ¦ <subscript-lst> , <input>

137 <call-lit> ::= <lit>

The name of the module to be called is saved for use in production 18.

138 <nn-lit> ::= <lit>

The literal string is placed into the constant area using an INT operator.

139 | SPACE

140 | QUOTE

141 <literal> ::= <nn-lit>

142 | <input>

The input value must be a numeric literal to be valid and is loaded into the constant area using an INT operator.

143 | ZERO

144 <lit/id> ::= <l/id>

145 | <nn-lit>

146 <program-id> ::= <id>

147 | <empty>

148 <read-id> ::= READ <id>

There are four read operations: RDF, RVL, RRS, and RRR.

149 <if-nonterminal>::=IF

III. NPS MICRO-COBOL INTERPRETER

A. GENERAL DESCRIPTION

The following sections describe the NPS MICRO-COBOL pseudo-machine in terms of the implementation, memory organization, interface actions and interpreter instructions. The pseudo-machine, which is constructed in the transient program area of CP/M, is the target machine for the compiler and is implemented through a programmed interpreter. The interpreter decodes each operation and either calls subroutines to perform the required actions or acts directly on the run time environment to control the actions of the interpreter. All communications between instructions is done through common areas in the program where information can be stored for later use. See figure [III-1] for an illustration of the pseudo-machine organization.

The machine contains a program counter and multiple parameter operations which contain all the information required to perform one complete action required by the language. Three eighteen digit, double length registers are used for arithmetic operations, along with a subscript stack used to compute subscript locations, a parameter stack to resolve the address of actual parameters and a set of flags which are used to pass branching information from one

instruction to another.

Addresses in the pseudo-machine are represented by 16 bit values. Any memory address greater than 20 hexidecimal is valid. Addresses less than 20 hexidecimal will be interpreted as having special significance. For example addresses one through eight are reserved for subscript stack references. All other addresses. in the machine are absolute addresses

The registers allow manipulation of signed numbers up to eighteen digits in length. Included in their representation is a sign indicator and the position of the assumed decimal point for the currently loaded number. Numbers are represented in standard COBOL "Display" or "Binary Coded Decimal" (COMP-3 or BCD) format. These numbers may have separate signs indicated by "+" and "-" or may have a "zone" indicator, denoting a negative sign, in the most significant byte of a number's storage location. Before operations occur on any number, it is converted to a packed decimal format and entered into one of the pseudo-machine registers.

B. MEMORY ORGANIZATION

The memory of the pseudo-machine is divided into three major areas: 1.) the data area is established by the DATA DIVISION statements of the source program, 2.) the constants area which is established by both the DATA and PROCEDURE

DIVISIONS of the source program, and 3.) the code area which is established by the PROCEDURE DIVISION.

The data area is the lowest area in the pseudo-machine. This area contains the storage for identifiers declared in the DATA DIVISION. Additionally, the data area contains the File Control Block (FCB) and the buffer space (128 bytes) for all files declared in the source program.

Immediately following the data area is the code area. This contiguous area of storage contains all executable code generated. The constants area is located in high memory of the pseudo-machine. This area contains all edit field masks as well as all numeric and non-numeric literals. Figure [III-1] illustrates the memory organization of the pseudo-machine.

PSEUDO-MACHINE ORGANIZATION

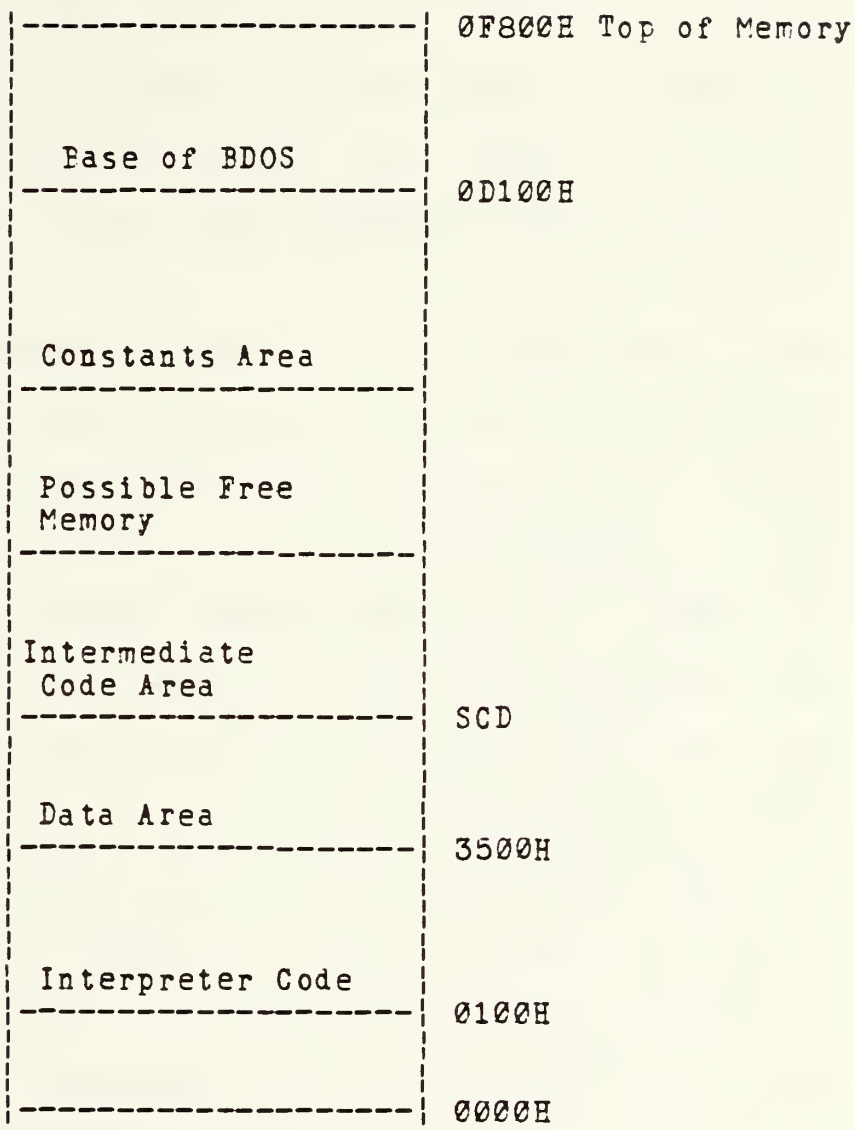


FIGURE III-1

C. INTERPRETER INTERFACE

The interpreter consists of two interface routines and the main interpreter program. To execute the interpreter the command EXEC <filename>, (where file type is CIN), is typed at the terminal. This action causes the two interface routines, BUILD and INTRDR, to be brought into memory. See figure [III-2] which illustrates the memory organization immediately after BUILD and INTRDR have been copied into memory.

The BUILD routine reads in the intermediate code, initializes all memory locations requiring initialization, and resolves all unresolved address references. In addition the BUILD routine loads subroutines into memory. If a SBR instruction is encountered during execution of BUILD, the SUB$FLAG is set as an indicator that subroutines will have to be loaded. The name of the subroutine is saved and when the TER instruction is encountered a check of the SUB$FLAG is made and if set each subroutine is loaded into memeory. A table similar to the compiler's symbol table is used to maintain the names, location, and status (loaded or unloaded) of each subroutine. Until a subprogram is loaded the actual branch address is not known. The same mechanism used for resolving forward branches to paragraphs is used to backstuff all previous references to the called procedure. Once loaded the address is known so no futher action is

required. See figure [III-5] for an illustration of a subroutine table entry.

The INTRDR routine reads the interpreter program into memory and transfers control to it.

The intermediate code instructions fall into two categories: 1.) instructions used by BUILD to establish the run time environment and, 2.) instructions to be executed by the interpreter. The following four instructions are generated in the compiler for use by the BUILD routine; SCD, INT, BST, and TER.

The SCD (start code) instruction is the last instruction generated by PART ONE and indicates where the first executable instruction for the intermediate code is to be loaded. This corresponds to the address immediately following the data area in the pseudo-machine. See Figure [III-1] which illustrates the relative location of the address that is associated with the SCD instruction. Figure [III-4] illustrates the memory organization of the pseudo-machine when subroutines are used.

MEMORY ORGANIZATION AFTER BUILD AND INTRDR
HAVE BEEN LOADED INTO MEMORY

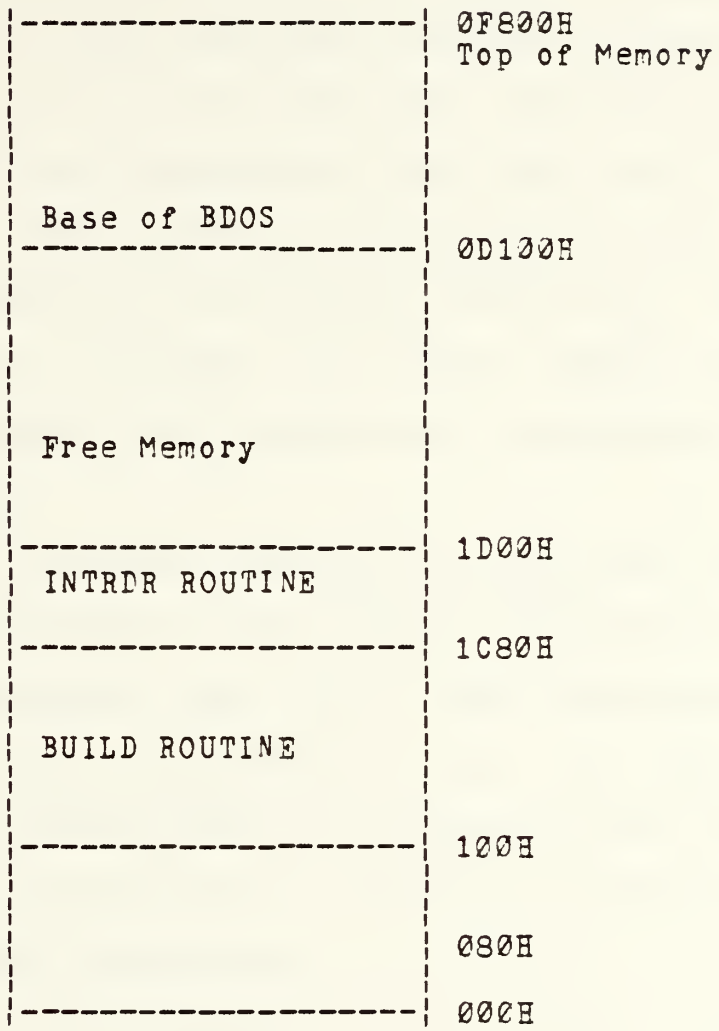


FIGURE III-2

The INT (initialize) instruction causes the BUILD routine to initialize the data area with the values associated with those identifiers in the DATA DIVISION of the source program that had VALUE CLAUSES. In addition, the INT instruction causes the BUILD routine to initialize the constants area with all the edit masks for those identifiers of the numeric and alphanumeric edit type, and all literals encountered in the PROCEDURE DIVISION of the source program.

The BST (backstuff) instruction resolves all unresolved references, i.e. branches to labels defined after the respective PERFORM or GO statement was encountered in the source program.

The TER (terminate) instruction is the last instruction generated by PART TWO of the compiler and indicates the end of the intermediate code file. Upon encountering a TER instruction in the intermediate code the BUILD routine inserts a STP instruction in its place. The STP instruction will cause the interpreter to terminate interpretation of the program when encountered.

All other code generated by the compiler is copied into the code area of the pseudo-machine by the BUILD routine. See Figure [III-3] for an illustration of the memory organization at this point in the initialization routine. The final action taken by the BUILD routine is to move the INTRDR routine into the input buffer at 80H and transfer control to it. This frees the area from 100H to the base of

the data area for the interpreter.

The INTRDR routine reads the interpreter program into memory starting at 100H and transfers control to it. From this point on the interpreter program executes the intermediate code that was loaded into the pseudo-machine.

MEMORY ORGANIZATION AFTER INTERMEDIATE CODE IS LOADED INTO MEMORY AND BEFORE THE INTERPRETER IS LOADED

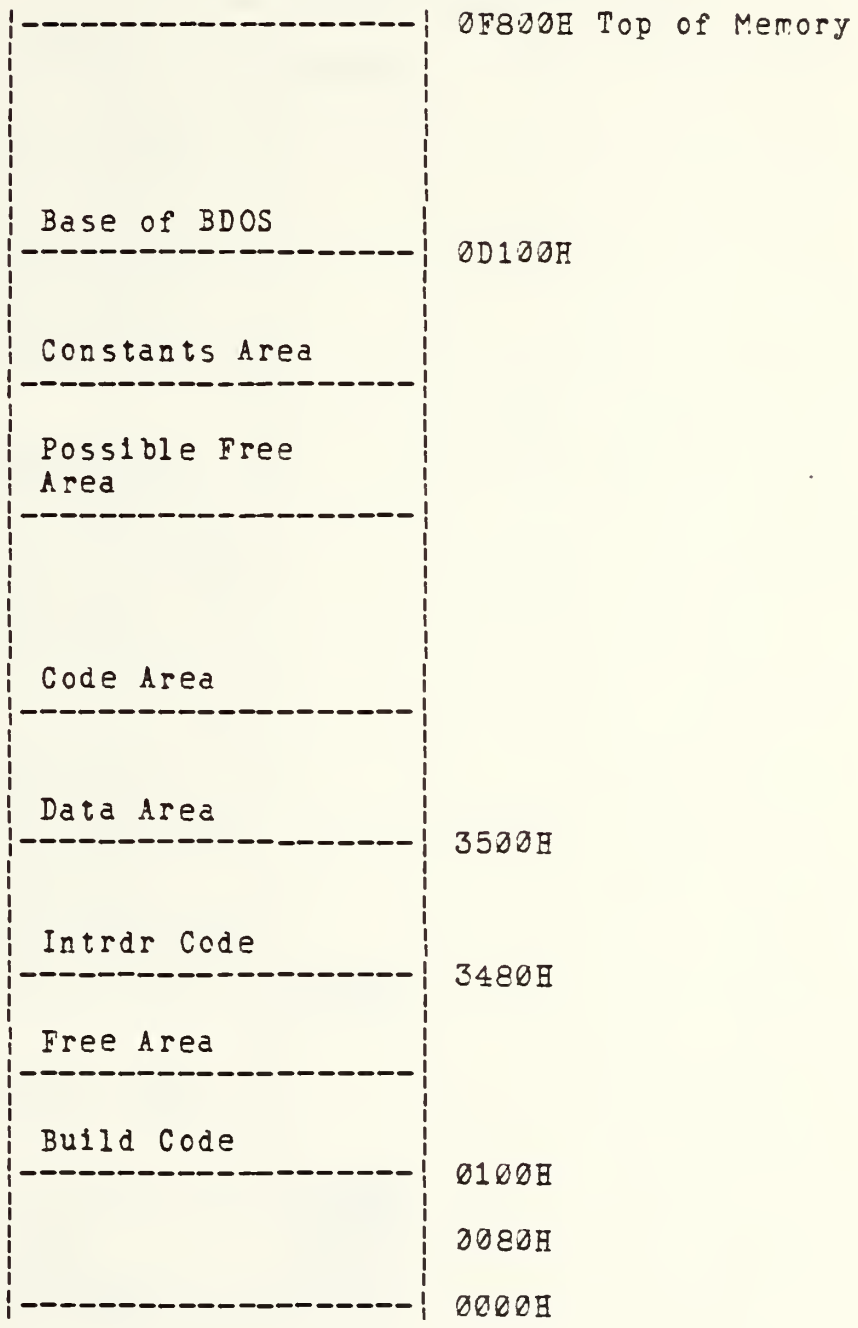
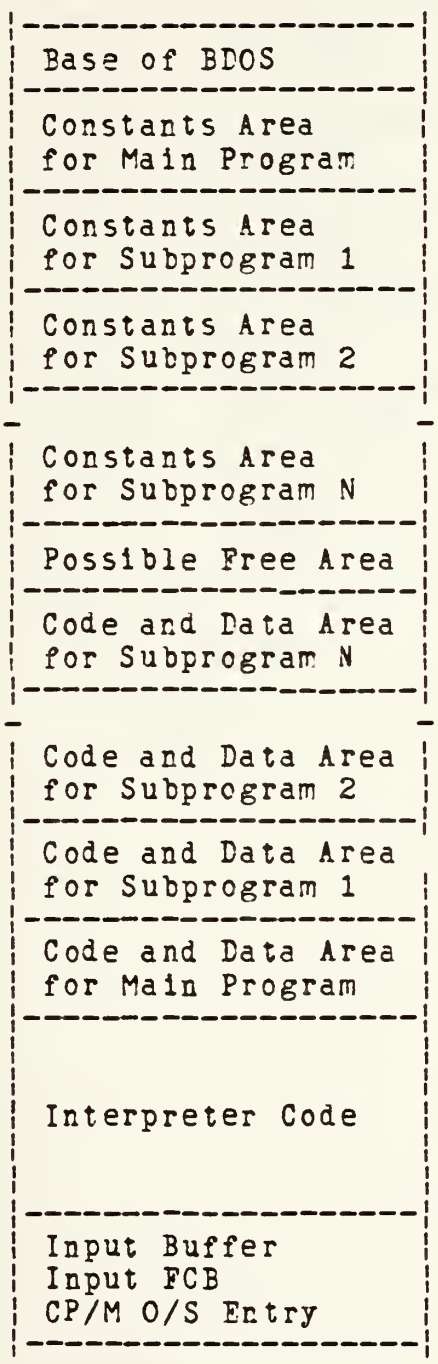


FIGURE III-3

MEMORY ORGANIZATION AFTER THE INTERMEDIATE CODE,
SUBROUTINES AND THE INTERPRETER ARE LOADED.


Base of BDOS
Constants Area for Main Program
Constants Area for Subprogram 1
Constants Area for Subprogram 2
Constants Area for Subprogram N
Possible Free Area
Code and Data Area for Subprogram N
Code and Data Area for Subprogram 2
Code and Data Area for Subprogram 1
Code and Data Area for Main Program
Interpreter Code
Input Buffer
Input FCB
CP/M O/S Entry

0F800H Top of Memory
0D00H
3500H
0100H
0080H
005CH
0000H

FIGURE III-4

SUBPROGRAM TABLE ENTRY

BYTE	SUBPROGRAM TABLE ENTRY
0-1	collision link (00 00)
2-3	subprogram address (48 52)
4-5	low$offset (00 00)
5-6	high$offset (00 00)
7-14	file name (49 43 31 35 32 20 20 20)
15	load$flag (00)

CALL 'IC152'

FIGURE III-5

D. PSEUDO-MACHINE INSTRUCTIONS

This section briefly covers the pseudo-machine instructions used in the interpreter, their format, and the actions which they accomplish.

1. Format

All of the interpreter instructions consist of an instruction number followed by a list of parameters. The following sections describe the instructions, list the required parameters, and describe the actions taken by the machine in executing each instruction. In each case, parameters are denoted informally by the parameter name enclosed in brackets. The BRN branching instruction, for example, uses the single parameter <branch address> which is the target of the unconditional branch.

As each instruction number is fetched from memory, the program counter is incremented by one. The program counter is then either incremented to the next instruction number, or a branch is taken.

The three eighteen digit registers which are used by the instructions covered in the following sections are referred to as registers zero, one, and two.

2. Arithmetic Operations

There are five arithmetic instructions which act upon the three registers. In all cases, the result is

Placed in register two. Operations are allowed to destroy the input values during the process of creating a result, therefore, a number loaded into a register is not available for a subsequent operation.

ADD: (addition). Sum the contents of register zero and register one.

Parameters: no parameters are required.

SUB: (subtract). Subtract register zero from register one.

Parameters: no parameters are required.

MUL: (multiply). Multiply register zero by register one.

Parameters: no parameters are required.

DIV: (divide). Divide register one by the value in register zero. The remainder is not retained.

Parameters: no parameters are required

RND:(round). Round register two to the last significant decimal place.

Parameters: no parameters are required.

3. Branching

The machine contains the following flags which are used by the conditional instructions in this section.

BRANCH flag -- indicates if a branch is to be taken;

END OF RECORD flag -- indicates that an end of input condition has been reached when an attempt was made

to read input;

OVERFLOW flag -- indicates the loss of information from a register due to a number exceeding the available size;

INVALID flag -- indicates an invalid action in writing to a direct access storage device.

All of the branch instructions are executed by changing the value of the program counter. Some are unconditional branches and some test for condition flags which are set by other instructions. A conditional branch is executed by testing the branch flag which is initialized to false. A true value causes a branch by changing the program counter to the value of the branch address. The branch flag is then reset to false. A false value causes the program counter to be incremented to the next sequential instruction.

BRN: (branch to an address). Load the program counter with the <branch address>.

Parameters: <branch address>

The next three instructions share a common format. The memory field addressed by the <memory address> is checked for the <address length>, and if all the characters match the test condition, the branch flag is complimented

Parameters: <memory address> <address length> <branch address>

CAL: (compare alphabetic). Compare a memory field

for alphabetic characters.

CNS: (compare numeric signed). Compare a field for numeric characters allowing for a sign character.

CNU: (compare numeric unsigned). Compare a field for numeric characters only.

DEC: (decrement a counter and branch if zero). Decrement the value of the <address counter> by one; if the result is zero before or after the decrement, the program counter is set to the <branch address>. If the result is not zero, the program counter is incremented by four.

Parameters: <address counter> <branch address>

EOR: (branch on END OF RECORD flag). If the END OF RECORD flag is true, it is set to false and the program counter is set to the <branch address>. If false, the program counter is incremented by two.

Parameters: <branch address>

GDP: (go to - depending on). The memory location addressed by the <number address> is read for the number of bytes indicated by the <memory length>. This number indicates which of the <branch addresses> is to be used. The first parameter is a bound on the number of branch addresses. If the number is within the range, the program counter is set to the indicated address. An out-of-bounds value causes the program counter to be advanced to the next sequential instruction.

Parameters: <bound number - byte> <memory length> <memory

address> <branch addr-1> <branch addr-2> ... <branch addr-n>

INV: (branch if INVALID flag true). If the invalid-file-action flag is true, then it is set to false, and the program counter is set to the branch address. If it is false, the program counter is incremented by two.

Parameters: <branch address>

PER: (perform). The code address addressed by the <change address> is loaded with the value of the <return address>. The program counter is then set to the <branch address>.

Parameters: <branch address> <change address> <return address>

RET: (return). If the value of the <branch address> is not zero, then the program counter is set to its value, and the <branch address> is set to zero. If the <branch address> is zero, the program counter is incremented by two.

Parameters: <branch address>

REQ: (register equal). This instruction checks for a zero value in register two. If it is zero, the branch flag is complemented. A conditional branch is taken.

Parameters: <branch address>

RGT: (register greater than). Register two is checked for a negative sign. If present, the branch flag is complemented. A conditional branch is taken.

Parameters: <branch address>

RLT: (register less than). Register two is checked for a positive sign, and if present, the branch flag is complemented. A conditional branch is taken.

Parameters: <branch address>

SER: (branch on size error). If the overflow flag is true, then the program counter is set to the branch address, and the overflow flag is set to false. If it is false, then the program counter is incremented by two.

Parameters: <branch address>

The next three instructions are of similar form in that they compare two strings and set the branch flag if the condition is true.

Parameters: <string addr-1> <string addr-2> <length - address> <branch address>

SEQ: (strings equal). The condition is true if the strings are equal.

SGT: (string greater than). The condition is true if string one is greater than string two.

SLT: (string less than). The condition is true if string one is less than string two.

4. Moves

The machine supports a variety of move operations for various formats and types of data. It does not support direct moves of numeric data from one memory field to another Instead, all numeric moves go through the registers.

The next seven instructions perform the same function. They load a register with a numeric value and differ only in the type of number that they expect to see in memory at the <number address>. All seven instructions cause the program counter to be incremented by five. Their common format is given below.

Parameters: <number address> <byte length> <byte decimal count> <byte register to load>

LOD: (load literal). Register two is loaded with a constant value. The decimal point indicator is not set in this instruction. The literal will have an actual decimal point in the string if required.

LD1: (load numeric). Load a numeric field.

LD2: (load postfix numeric). Load a numeric field with an internal trailing sign.

LD3: (load prefix numeric). Load a numeric field with an internal leading sign.

LD4: (load separated postfix numeric). Load a numeric field with a separate leading sign.

LD5: (load separated prefix numeric). Load a numeric field with a separate trailing sign.

LD6: (load packed numeric). Load a packed numeric field.

MED: (move into alphanumeric edited field). The edit mask is loaded into the <to address> to set up the move, and then the <from address> information is loaded. The

program counter is incremented by ten.

Parameters: <to address> <from address> <length of move address> <edit mask address> <edit mask length, address>

MNE: (move into a numeric edited field). First the edit mask is loaded into the receiving field, and then the information is loaded. Any decimal point alignment required will be performed. Truncation of significant digits will not set the overflow flag. The program counter is incremented by twelve.

Parameters: <to address> <from address> <address length of move> <edit mask address> <address mask length> <byte to decimal count> <byte from decimal count>

MOV: (move into an alphanumeric field). The memory field given by the <to address> is filled by the from field for the <move length> and then filled with blanks in the following positions for the <fill count>.

Parameters: <to address> <from address> <address move length> <address fill count>

STI: (store immediate register two). The contents of register two are stored into register zero and the decimal count and sign indicators are set.

Parameters: none.

The store instructions are grouped in the same order as the load instructions. Register two is stored into memory at the indicated location. Alignment is performed and any truncation of leading digits causes the overflow

flag to be set. All six of the store instructions cause the program counter to be incremented by four. The format for these instructions is as follows.

Parameters: <address to store into> <byte length> <byte decimal count>

ST0: (store numeric). Store into a numeric field.

ST1: (store postfix numeric). Store into a numeric field with an internal trailing sign.

ST2: (store prefix numeric). Store into a numeric with an internal leading sign.

ST3: (store separated postfix numeric). Store into a numeric field with a separate trailing sign.

ST4: (store separated prefix numeric). Store into a numeric field with a separate leading sign.

ST5: (store packed numeric). Store into a packed numeric field.

5. Input-Output

The following instructions perform input and output operations. Files are defined as having the following characteristics: they are either sequential or random and, in general, files created in one mode are not required to be readable in the other mode. Standard files consist of fixed length records, and variable length files need not be readable in a random mode. Further, there must be some character or character string that delimits a variable

length record.

ACC: (accept). Read from the system input device into memory at the location given by the <memory address>. The program counter is incremented by three.

Parameters: <memory address> <byte length of read>

CLS: (close). Close the file whose file control block is addressed by the <fcb address>. The program counter is incremented by two.

Parameters: <fcb address>

DIS: (display). Print the contents of the data field pointed to by <memory address> on the system output device for the indicated length and advance the line output if <flag> is set. The program counter is incremented by four.

Parameters: <memory address> <byte length> <flag>

There are three open instructions with the same format. In each case, the file defined by the file control block referenced will be opened for the mode indicated. The program counter is incremented by two.

Parameters: <fcb address>

OPN: (open a file for input).

OP1: (open a file for output).

OP2: (open a file for both input and output). This is only valid for files on a random access device.

The following file actions all share the same format. Each performs a file action on the file referenced by the file control block. The record to be acted upon is

given by the <record address>. The program counter is incremented by six.

Parameters: <FCB address> <record address> <record length - address>.

DLS: (delete a record from a sequential file). Remove the record that was just read from the file. The file is required to be open in the input-output mode.

RDF: (read a sequential file). Read the next record into the memory area.

WTF: (write a record to a sequential file). Append a new record to the file.

RVL: (read a variable length record).

WVL: (write a variable length record).

RWS: (rewrite sequential). The rewrite operation writes a record from memory to the file, overlaying the last record that was read from the device. The file must be open in the input-output mode.

The following file actions require random files rather than sequential files. They make use of a random file pointer which consists of a <relative address> and a <relative length>. The memory field holds the number to be used in disk operations or contains the relative record number of the last disk action. The relative record number is an index into the file which addresses the record being accessed. After the file action, the program counter is incremented by nine.

Parameters: <FCB address> <record address> <record length - address> <relative address> <relative length - byte>.

DLR: (delete a random record). Delete the record addressed by the relative record number.

RRR: (read random relative). Read a random record relative to the record number.

RRS: (read random sequential). Read the next sequential record from a random file. The relative record number of the record read is loaded into the memory reference.

RWR: (rewrite a random record).

WRR: (write random relative). Write a record into the area indicated by the memory reference.

WRS: (write random sequential). Write the next sequential record to a random file. The relative record number is returned.

6. Subroutine Instructions

The next three instructions are used to transfer control to a subroutine and pass the location of formal parameters.

EXT: (exit subroutine). The program counter is set to the last value on the return stack and the actual partameters on the parameter stack are removed revealing any parameters that may be needed in the calling procedure.
Parameters: No parameters are required.

SBR: (call a subroutine). The program counter is

set to the beginning address of the called procedure. The return address is added to the return stack.

Parameters: <procedure name-8 bytes>

PAR: (parameter list). The parameters are added to the parameter stack.

Parameters: <number of parameters> <address parameter 1> <address parameter 2> .....

7. Special Instructions

The remaining instructions perform special functions required by the machine that do not relate to any of the previous groups.

NEG: (negate). Complement the value of the branch flag.

Parameters: No parameters are required.

LDI: (load a code address direct). Load the code address located five bytes after the LDI instruction with the contents of <memory address> after it has been converted to binary.

Parameters: <memory address> <length - byte>

SCR: (calculate a subscript). Load the subscript stack with the value indicated from memory. The address loaded into the stack is the <initial address> plus an offset. Multiplying the <field length> by the number in the <memory reference> gives the offset value.

Parameters: <initial address> <field length> <memory refer-

ence> <memory length> <stack level>

STD: (stop display). Display the indicated information and then terminate the actions of the machine. The operator is given a choice to allow the machine to continue or to terminate its actions.

Parameters: <memory address> <length - byte>

STP: (stop). Terminate the actions of the machine. The following instructions are actually instructions to the build program in setting up the machine enviromnent and are not used in the normal execution of the machine.

Parameters: no parameters are required.

BST: (backstuff). Resolve a reference to a label. Labels may be referenced prior to their definition, requiring a chain of resolution addresses to be maintained in the code. The latest location to be resolved is maintained in the symbol table and a pointer at that location indicates the next previous location to be resolved. A zero pointer indicates no prior occurrences of the label. The code address referenced by <change address> is examined and if it contains zero, it is loaded with the <new address>. If it is not zero, then the contents are saved, and the process is repeated with the saved value as the change address after loading the <new address>.

Parameters: <change address> <new address>

INT: (initialize memory). Load memory with the <input string> for the given length at the <memory address>.

Parameters: <memory address> <address length> <input string>

SCD: (start code). Set the initial value of the program counter.

Parameters: <start address>

TER: (terminate). Terminate the initialization process and start executing code.

Parameters: no parameters are required.

IV. SYSTEM DEBUGGING METHODS AND TOOLS

A. DEBUGGING METHODOLOGY

Initial debugging began with implementation of key components of the compiler/interpreter that had prevented use of the Navy's ADPESO validation test programs. Additional work on the validation test programs was necassary to eliminate and/or correct minor errors within the test programs themselves. Once these errors were corrected the compiler/interpreter was able to compile and execute the ADPESO programs completely and an overall view of the problems and errors within the system was available for analysis.

Since compile time for each of the three main modules -- PART ONE, PART TWO, and INTERP -- took a mimimum of forty-five minutes, a step-wize refinement technique was employed. First the simplest problems were corrected all at the same time. Once this was accomplished the remaining problems were handled one at a time to prevent introducing new problems from side effects of the corrections. Debugging could then be confined to only one problem and side effects kept to a minimum. This technique required more compilations but it was felt that attempting to correct more than one problem at a time could cause severe side effects with an increase in overall debugging time.

B. INTERACTIVE TOOLS

Because the MICRO-COBOL compiler and interpreter were implemented under the CP/M operating system, the Symbolic Instruction Debugger [7], SID, which expands upon the features of the Dynamic Debugging Tool [8], DDT, was employed. Specifically, SID includes real-time breakpoints, fully monitored execution, symbolic disassembly, assembly, and memory display and fill functions. One feature which allowed the setting of breakpoints at actual memory locations corresponding to a program's source lines and symbolic names was used quite extensively. Another useful facility was the ability to display and alter the programs symbolic values, which enabled the substitution of values to check a proposed solution to an error.

C. CROSS REFERENCE LISTINGS

Another useful facility which eased the debugging effort was the cross reference listings produced by the PLM80 compiler used to compile the MICRO-COBOL compiler and interpreter. There were three different listings produced after each compilation: 1.) a line numbered source listing, 2.) a symbol address table, which included the name and actual memory address assigned for all symbols declared, and 3.) a line address table which cross referenced every line in the source listing with the 8080 code generated by the

PLM80 compiler for that particular line. These listings were almost indispensable with regard to testing and debugging, and their contribution cannot be overemphasized.

D. VALIDATION TESTS

The primary method for discovering errors was the HYPO-COBOL Compiler Validation System (HCCVS) Tape (from the Automated Data Processing Equipment Selection Office (ADPESO)). The transfer of these test programs from tape to a usable form on floppy diskettes was accomplished by Kiefer and Perry [14]. Additional errors were discovered through several additional test programs written to test areas that were not tested by the ADPESO programs or constructs that were not contained in the HYPO-COBOL specifications.

V. CONCLUSIONS AND RECOMMENDATIONS

The entire MICRO-COBOL Compiler/Interpreter has been tested, debugged and documented. The following specific language features and facilities previously not implemented, or implemented incorrectly, have been successfully implemented, tested and debugged during this project: 1.) the compiler's ability to handle any sequence of MICRO-COBOL language constructs (PIC CLAUSE, VALUE CLAUSE, OCCURS CLAUSE, and USAGE COMP-3 CLAUSE) in the declaration of an identifier, 2.) record identifier declarations with up to ten levels of elementary field items, 3.) record and elementary field identifier redefinitions, 4.) nested redefinitions, and 5.) error message generation for duplicate identifier declarations within the DATA DIVISION, rework of the BCD arithemetic package including the ROUND and SIZE ERROR options, 7.) implementation of the Move Numeric Edited command, 8.) implementation of nested IF-THEN-ELSE statements, 9.) implementation of the PERFORM VARYING clause, 10.) modification of all MOVE commands, 11.) modification of the EXIT clause for use with subroutines, 12.) modification of the STOP DISPLAY clause to allow operator restart, 13.) implementation of subroutines including the CALL, USING and LINKAGE SECTION clauses, 14.) modification of the WRITE BEFORE/AFTER clause, 15).

implementation of COMP-3 and SIGN LEADING/TRAILING options, 16.) addition of the list and code compiler toggles to include a list file with errors and line numbers and the capability of surpressing code generation for rapid syntax checking, and 17) expansion of the grammar to include the COMPUTE verb, the logical operators "AND" and "OR", indexed files, and the relational operators "<", ">", and "=".

NPS MICRO-COBOL compiles at a rate of approximately 500 lines per minute using a Z-80 microprocessor with a 4MHZ clock on a standard eight inch floppy diskette. With the use of optional toggles such as NO$CODE or NO$LIST compilation rate increases to approximately 700 lines per minute and a maximum rate of approximately 900 lines per minute with both NO$CODE and NO$LIST toggles selected. Memory usage is kept to a minimum through the use of overlays thus allowing fairly complex COBOL programs to be written and executed on a modest size microcomputer system. The present development system is designed to run in only 48K of main memory and can run in as little as 20K or as much as the 64K maximum address space of an 8080 or Z-80 microcomputer. These two features in addition to clear error diagnostics make the NPS MICRO-COBOL compiler/interpreter an excellent tool for teaching introductory COBOL programming.

NPS MICRO-COBOL has been validated by the complete ADPESO validation test package for HYPO-COBOL. In addition to the twenty-five test programs from that package, several

test programs designed to test the additional features implemented which were not in HYPO-COBOL and several application programs have been compiled and executed to the sum of approximately 50,000 lines of COBOL code.

In addition, the NPS MICRO-COBOL compiler documentation has been updated. This documentation includes the following: 1.) module organization, 2.) module interfaces, 3.) memory organization of the Interpreter, 4.) construction and data initialization of the symbol table, and 5.) key internal data structures.

Several areas remain which could be implemented to enhance the NPS MICRO-COBOL compiler/interpreter system, these include: 1.) implementation of the COMPUTE verb, 2.) implementation of multiple Open's, and Close's, 3.) implementation of multi-dimensional tables, 4.) implementation of the logical operators "AND" and "OR", and 5.) implementation of the optional comparison operators "<", ">", and "=".

APPENDIX A

NPS MICRO-COBOL USER'S MANUAL

VERSION 2.0

TABLE OF CONTENTS

I. ORGANIZATION 103

II. MICRO-COBOL ELEMENTS 104

III. COMPILER TOGGLES 144

IV. RUN TIME CONVENTIONS 146

V. FILE INTERACTIONS WITH CP/M 148

VI. ERROR MESSAGES 150

A. COMPILER FATAL MESSAGES 150

B. COMPILER WARNINGS 150

C. INTERPRETER FATAL ERRORS 154

D. INTERPRETER WARNING MESSAGES 155

I. ORGANIZATION

The compiler is designed to run on an 8080 system in an interactive mode through the use of a teletype or console. It requires at least 24K of main memory and a mass storage device for reading and writing. The compiler is composed of two parts , each of which reads a portion of the input file. Part One reads the input program to the end of the Data Division and builds the symbol table. At the end of the Data Division, Part One is overlayed by Part Two which uses the symbol table to produce the code. The output code is written as it is produced to minimize the use of internal storage.

The EXEC Program builds the core image for the intermediate code and performs such functions as backstuffing addresses and offsetting address in subroutines. EXEC then copies the interpreter(CINTERP.COM) into memory and transfers control to the it. The interpreter is controlled by a large case statement that decodes the instructions and performs the required actions.

II. MICRO-COBOL ELEMENTS

This section contains a description of each element in the language and shows simple examples of their use. The following conventions are used in explaining the formats: Elements enclosed in broken braces < > are themselves complete entities and are described elsewhere in the manual. Elements enclosed in braces { } are choices, one of the elements which is to be used. Elements enclosed in brackets [ ] are optional. All elements in capital letters are reserved words and must be spelled exactly.

User names are indicated in lower case. These names are unrestricted in length, however they must be unique within the first 15 characters. The only other restriction on user names is that the first character must be an alpha character. The remainder of the user name can have any combination of representable characters in it.

The input to the compiler does not need to conform to standard COBOL format. Free form input will be accepted as the default condition. If desired, sequence numbers can be entered in the first six positions of each line. However, a toggle needs to be set to cause the compiler to ignore the sequence numbers.

The first character position on any line is used to indicate the following:-

* - indicates a comment entry.

: - indicates a debugging line.

/ - indicates a page eject.

ELEMENT:

IDENTIFICATION DIVISION Format

FORMAT:

```
IDENTIFICATION DIVISION.

PROGRAM-ID. <comment>.

[AUTHOR. <comment>.]

[DATE-WRITTEN. <comment>.]

[SECURITY. <comment>.]
```

DESCRIPTION:

This division provides information for program identification for the reader. The order of the lines is fixed.

EXAMPLES:

```
IDENTIFICATION DIVISION.

PROGRAM-ID. SAMPLE.

AUTHOR. HAL R POWELL.
```

ELEMENT:

ENVIRONMENT DIVISION Format

FORMAT:

```
[ ENVIRONMENT DIVISION.
  CONFIGURATION SECTION.
  SOURCE-COMPUTER. <comment> [DEBUGGING MODE].
  OBJECT-COMPUTER. <comment>.
  [INPUT-OUTPUT SECTION.
  FILE-CONTROL.
     <file-control-entry> . . .
  [I-O-CONTROL.
     SAME file-name-1 file-name-2 [file-name-3]
             [file-name-4] [file-name-5].  ]  ]  ]
```

DESCRIPTION:

This division determines the external nature of a file. In the case of CP/M all of the files used can be accessed either sequentially or randomly except for variable length files which are sequential only. The

debugging mode is also set by this section. The DEBUGGING MODE clause is used in conjunction with the ':' to indicate conditional compilation. If this clause is specified all debugging lines (those with a ':' in column one) are compiled. If this clause is not specified, all debugging lines are treated as comments. In addition the DEBUGGING MODE can be specified by using the compiler toggle 'D'.

ELEMENT:

<file-control-entry>

FORMAT:

1.

```
SELECT file-name
    ASSIGN implementor-name
    [ORGANIZATION SEQUENTIAL]
    [ACCESS SEQUENTIAL].
```

2.

```
SELECT file-name
    ASSIGN implementor-name
    ORGANIZATION RELATIVE
    [ACCESS {SEQUENTIAL [RELATIVE data-name]}].
            {RANDOM RELATIVE data-name      }
```

3.

```
SELECT file-name
```

```
ASSIGN implementor-name

ORGANIZATION INDEXED

[ACCESS {SEQUENTIAL}].

       {RANDOM    }
```

DESCRIPTION:

The file-control-entry defines the type of file that the program expects to see. There is no difference on the diskette, but the type of reads and writes that are performed will differ. For CP/M the implementor name needs to conform to the normal specifications. Indexed is not implemented.

EXAMPLES:

```
SELECT CARDS

   ASSIGN CARD.FIL.

SELECT RANDOM-FILE

   ASSIGN A.RAN

   ORGANIZATION RELATIVE

   ACCESS RANDOM RELATIVE RAND-FLAG.
```

ELEMENT:

DATA DIVISION Format

FORMAT:

```
DATA DIVISION.

[FILE SECTION.

[FD file-name

   [BLOCK integer-1 RECORDS]

   [RECORD [integer-2 TO] integer-3]

   [LABEL RECORDS {STANDARD}]

                  {OMITTED }

   [VALUE OF implementor-name-1 literal-1

       [implementor-name-2 literal-2] ... ].

[<record-description-entry>] ...] ...

[WORKING-STORAGE SECTION.

[<record-description-entry>] ... ]

[LINKAGE SECTION.
```

[<record-description-entry>] ...]

DESCRIPTION:

This is the section that describes how the data is structured. There are no major differences from standard COBOL except for the following: 1. Label records make no sense on the diskette so no entry is required. 2. The VALUE OF clause likewise has no meaning for CP/M. If a record is given two lengths as in RECORD 12 TO 128, the file is taken to be variable length and can only be accessed in the sequential mode. See the section on files for more information.

ELEMENT:

<comment>

FORMAT:

any string of characters

DESCRIPTION:

A comment is a string of characters. It may include anything other than a period followed by a blank or a reserved word, either of which terminate the string. Comments may be empty if desired, but the terminator is still required by the program.

EXAMPLES:

this is a comment

anotheroneallruntogether

8080b 16K

ELEMENT:

<data-description-entry> Format

FORMAT:

```
level-number {data-name}
             {FILLER   }
[REDEFINES data-name]
[PIC character-string]
[USAGE {COMP   }]
       {COMP-3}
       {COMPUTATIONAL}
       {DISPLAY}
[SIGN {LEADING} [SEPARATE]]
      {TRAILING}
[OCCURS integer]
[SYNC [LEFT ]]
      [RIGHT]
```

[VALUE literal].

DESCRIPTION:

This statement describes the specific attributes of the data. Since the 8080 is a byte machine, there was no meaning to the SYNC clause, and thus it has not been implemented, however existing programs that are transfered to MICRO-COBOL and use this feature will compile and execute successfully. All numeric data are maintained in DISPLAY format or packed BCD if the COMP-3 option is used.

EXAMPLES:

```
01 CARD-RECORD.

   02 PART PIC X(5).

   02 NEXT-PART PIC 99V99 USAGE DISPLAY.

   02 FILLER.

      03 NUMB PIC S9(3)V9 SIGN LEADING SEPARATE.

      03 LONG-NUMB 9(15).

      03 STRING REDEFINES LONG-NUMB PIC X(15).

   02 ARRAY PIC 99 OCCURS 100.
```

ELEMENT:

PROCEDURE DIVISION Format

FORMAT:

1.

```
PROCEDURE DIVISION [USING name1 [name2] ... [name5]].

section-name SECTION.

[paragraph-name. <sentence> [<sentence> ... ] ... ] ...
```

2.

```
PROCEDURE DIVISION [USING name1 [name2] ... [name5]].

paragraph-name. <sentence> [<sentence> ...] ...
```

DESCRIPTION:

As is indicated, if the program is to contain sections, then the first paragraph must be in a section.

ELEMENT:

<sentence>

FORMAT:

<imperative-statement>

<conditional-statement>

ELEMENT:

<imperative-statement>

FORMAT:

The following verbs are always imperatives:

ACCEPT

CALL

CLOSE

DISPLAY

EXIT

GO

MOVE

OPEN

PERFORM

STOP

The following may be imperatives:
arithmetic verbs without the SIZE ERROR statement
and DELETE, WRITE, and REWRITE without the INVALID option.

<conditional-statements>

ELEMENT:

<conditional-statements>

FORMAT:

IF

READ

arithmetic verbs with the SIZE ERROR statement

and DELETE, WRITE, and REWRITE with the INVALID option.

ELEMENT:

ACCEPT

FORMAT:

ACCEPT <identifier>

DESCRIPTION:

This statement reads up to 255 characters from the console. The usage of the item must be DISPLAY.

EXAMPLES:

```
ACCEPT IMMAGE.

ACCEPT NUM(9).
```

ELEMENT:

ADD

FORMAT:

```
ADD {identifier-1} [{identifier-2 }] ...  TO identifier-m
    {literal-1   }  {literal-2    }

   [ROUNDED] [SIZE ERROR <imperative-statement>]
```

DESCRIPTION:

This instruction adds either one number to a second with the result being placed in the last location. Multiple adds have not been implemented.

EXAMPLES:

```
ADD 10 TO NUMB1

ADD X TO Z ROUNDED.

ADD 100 TO NUMBER SIZE ERROR GO ERROR-LOC
```

ELEMENT:

CALL

FORMAT:

CALL literal [USING name1 [name2] ... [nameN]]

DESCRIPTION:

Control is transfered to the called procedure with an address of each of the parameters to be passed. The parameters map to those in the linkage section of the called program. The type and size of the parameters must match exactly.

EXAMPLES:

```
CALL 'NC152' USING DN1

CALL 'PRINT'

CALL 'ADDLIST' USING VAR1 VAR2 VAR3
```

ELEMENT:

CLOSE

FORMAT:

CLOSE file-name

DESCRIPTION:

Files must be closed if they have been written. However, the normal requirement to close an input file prior to the end of processing does not exist.

EXAMPLES:

```
CLOSE FILE1

CLOSE RANDFILE
```

ELEMENT:

DELETE

FORMAT:

DELETE file-name [INVALID <imperative-statement>]

DESCRIPTION:

This statement requires the file-name of the item to be deleted. The record is logically removed by filling it with a high value character, which is not displayable to the console or line printer. The logical record space can be used again by writing a valid record in its place.

EXAMPLES:

DELETE FILE-NAME

ELEMENT:

DISPLAY

FORMAT:

```
DISPLAY {identifier} [{identifier-1}] . . . [{identifier-N}]
        {literal   } {literal-1   } . . . {literal-N   }
```

DESCRIPTION:

This displays the contents of an identifier or displays a literal on the console. Usage must be DISPLAY. The maximum length of the display is 80 characters for literal values and 255 characters for identifiers.

EXAMPLES:

```
DISPLAY MESSAGE-1

DISPLAY MESSAGE-3 10

DISPLAY 'THIS MUST BE THE END'
```

ELEMENT:

DIVIDE

FORMAT:

```
DIVIDE {identifier} INTO identifier-1 [ROUNDED]
       {literal   }
   [SIZE ERROR <imperative-statement>]
```

DESCRIPTION:

The result of the division is stored in identifier-1; any remainder is lost.

EXAMPLES:

```
DIVIDE NUMB INTO STORE

DIVIDE 25 INTO RESULT
```

ELEMENT:

EXIT

FORMAT:

EXIT [PROGRAM]

DESCRIPTION:

The EXIT command causes no action by the interpreter but allows for an empty paragraph for the construction of a common return point. The optional PROGRAM terminates a subroutine and returns to the calling program. It's use in the main program couses no action to be taken.

EXAMPLES:

EXIT PROGRAM

EXIT

ELEMENT:

GO

FORMAT:

1.

GO procedure-name

2.

GO procedure-1 [procedure-2] ... procedure-20

DEPENDING identifier

DESCRIPTION:

The GO command causes an unconditional branch to the routine specified. The second form causes a forward branch depending on the value of the contents of the identifier. The identifier must be a numeric integer value. There can be no more than 20 procedure names.

EXAMPLES:

GO READ-CARD.

```
GO READ1 READ2 READ3 DEPENDING READ-INDEX.
```

ELEMENT:

IF

FORMAT:

```
IF <condition> {stmt-lst    } END-IF

IF <condition> {stmt-lst    } ELSE {stmt-lst} END-IF

               {NEXT SENTENCE}      {NEXT SENTENCE}
```

DESCRIPTION:

This is an enhanced version of the standard COBOL IF statement. Nesting of IF statements is allowed.

EXAMPLES:

```
IF A GREATER B ADD A TO C ELSE GO ERROR-ONE END-IF.

IF A NOT NUMERIC NEXT SENTENCE ELSE MOVE ZERO TO A END-IF.

IF A LESS B
   DISPLAY A
   DISPLAY B END-IF.

IF A GREATER B
   DISPLAY A
   DISPLAY B
```

```
ELSE
   DISPLAY C
   DISPLAY D END-IF.
IF A GREATER B
   IF A GREATER C
      DISPLAY A
   ELSE
      DISPLAY C
   END-IF
ELSE
   IF B GREATER C
      DISPLAY B
   ELSE
      DISPLAY C
   END-IF
END-IF.
```

ELEMENT:

MOVE

FORMAT:

```
MOVE {identifier-1} TO identifier-2
     {literal     }
```

DESCRIPTION:

The standard list of allowable moves applies to this action. As a space saving feature of this implementation, all numeric moves go through the accumulators. This makes numeric moves slower than alpha-numeric moves, and where possible they should be avoided. Any move that involves picture clauses that are exactly the same can be accomplished as an alpha-numeric move if the elements are redefined as alpha-numeric; also all group moves are alpha-numeric.

EXAMPLES:

```
MOVE SPACE TO PRINT-LINE.

MOVE A(10) TO B(PTR).
```

ELEMENT:

MULTIPLY

FORMAT:

```
MULTIPLY {identifier} BY identifier-2 [ROUNDED]
         {literal    }

   [SIZE ERROR <imperative-statement>]
```

DESCRIPTION:

The multiply routine uses a double length register to calculate the result. This allows the result generated to be of maximum precision. The actual value stored will be determined by the amount of storage allocated for the variable. Overflow will occur if the number in the register is larger than the variable. If the precision in the register is greater than the variable trucation occurs unless the round option is specified.

EXAMPLES:

```
MULTIPLY X BY Y.

MULTIPLY A BY B(7) SIZE ERROR GO OVERFLOW.
```

ELEMENT:

OPEN

FORMAT:

```
OPEN {INPUT file-name-1 } [{file-name-2}] ...

     {OUTPUT file-name-1} [{file-name-2}] ...

     {I-O file-name-1    } [{file-name-2}] ...
```

DESCRIPTION:

The three types of OPENS have exactly the same effect on the diskette. However, they do allow for internal checking of the other file actions. For example, a write to a file set open as input will cause a fatal error. Multiple opens have not been implemented.

EXAMPLES:

```
OPEN INPUT CARDS.

OPEN OUTPUT REPORT-FILE.
```

ELEMENT:

PERFORM

FORMAT:

1.

```
PERFORM procedure-name [THRU procedure-name-2]
```

2.

```
PERFORM procedure-name [THRU procedure-name-2]
   {identifier} TIMES
    {integer   }
```

3.

```
PERFORM procedure-name [THRU procedure-name-2]
   UNTIL <condition>
```

4.

```
PERFORM procedure-name VARYING {identifier}
   FROM {identifier} BY {identifier}
   UNTIL <condition>
```

DESCRIPTION:

All four options are supported. Branching may be either forward or backward, and the procedures called may have perform statements in them as long as the end points do not coincide or overlap.

EXAMPLES:

```
PERFORM OPEN-ROUTINE.

PERFORM TOTALS THRU END-REPORT.

PERFORM SUM 10 TIMES.

PERFORM SKIP-LINE UNTIL PG-CNT GREATER 60.

PERFORM REPEAT-AGAIN VARYING COUNTER FROM 1 BY 2
   UNTIL COUNTER EQUAL 10.
```

ELEMENT:

READ

FORMAT:

1.

READ file-name INVALID <imperative-statement>

2.

READ file-name END <imperative-statement>

DESCRIPTION:

The invalid condition is only applicable to files in a random mode. All sequential files must have an END statement.

EXAMPLES:

READ CARDS END GO END-OF-FILE.

READ RANDOM-FILE INVALID MOVE SPACES TO REC-1.

ELEMENT:

REWRITE

FORMAT:

REWRITE record-name [INVALID <imperative>]

DESCRIPTION:

REWRITE is only valid for files that are open in the I-O mode. The INVALID clause is only valid for random files. This statement results in the current record being written back into the place that it was just read from, the last executed read.

EXAMPLES:

REWRITE CARDS.

REWRITE RAND-1 INVALID PERFORM ERROR-CHECK.

ELEMENT:

STOP

FORMAT:

```
STOP {RUN    }
     {literal}
```

DESCRIPTION:

This statement stops execution of the program. If a literal is specified, then the literal is displayed on the console and a prompt is displayed giving the operator the option of terminating or continuing program execution.

EXAMPLES:

```
STOP RUN.

STOP 1.

STOP 'INVALID FINISH'.
```

For the last two examples the following prompt is displayed:

```
OPERATOR ENTER A <CR> TO CONTINUE
    OR ENTER AN "S" TO TERMINATE.
```

ELEMENT:

SUBTRACT

FORMAT:

```
SUBTRACT {identifier-1} [identifier-2] ... FROM identifier-m
         {literal-1   } [literal-2   ]
   [ROUNDED] [SIZE ERROR <imperative-statement>]
```

DESCRIPTION:

Identifier-m is decremented by the value of identifier/literal one. The results are stored back in identifier-m. Rounding and size error options are available if desired. Multiple subtracts have not been implemented.

EXAMPLES:

```
SUBTRACT 10 FROM SUB(12).

SUBTRACT A FROM C ROUNDED.
```

ELEMENT:

WRITE

FORMAT:

1.

```
WRITE record-name [{BEFORE} ADVANCING {INTEGER}]
                  {AFTER }           {PAGE   }
```

2.

```
WRITE record-name INVALID <imperative-statement>
```

DESCRIPTION:

The record specified is written to the file specified in the file section of the source program. The INVALID option only applies to random files.

EXAMPLES:

```
WRITE OUT-FILE.

WRITE RAND-FILE INVALID PERFORM ERROR-RECOV.
```

ELEMENT:

<condition>

FORMAT:

RELATIONAL CONDITION:

```
{identifier-1} [NOT] {GREATER} {identifier-2}
{literal-1}          {LESS   } {literal-2   }
                     {EQUAL  }
```

CLASS CONDITION:

```
identifier [NOT] {NUMERIC   }
                 {ALPHABETIC}
```

DESCRIPTION:

It is not valid to compare two literals. The class condition NUMERIC will allow for a sign if the identifier is signed numeric.

EXAMPLES:

```
A NOT LESS 10.

LINE GREATER 'C'.

NUMB1 NOT NUMERIC
```

ELEMENT:

Subscripting

FORMAT:

data-name (subscript)

DESCRIPTION:

Any item defined with an OCCURS may be referenced by a subscript. The subscript may be a literal integer, or it may be a data item that has been specified as an integer. If the subscript is signed, the sign must be positive at the time of its use.

EXAMPLES:

A(10)

ITEM(SUB)

III. COMPILER TOGGLES

There are six compiler toggles which are controlled by an entry following the compiler activation command, COBOL <filename>. The format of the entry consists of following <filename> by one space and then entering a "$" followed immediately by the desired toggles. There must be only one space after <filename> and no spaces between the "$" and the toggles. The following is an example of a typical entry:

COBOL EXAMPLE $S

This entry would cause the compiler to ignore the first six characters(used for sequence numbers) at the beginning of each input line. In each case the toggle reverses the default value.

$C -- No intermediate code. Default is off. Setting this toggle speeds initial compilation for syntax checking. When this toggle is set the "CIN" file is empty.

$D -- Debugging mode. Default is off. This toggle sets the debugging mode, which means all debugging lines(those with a ':' in column one) are compiled. If this toggle is not set and the DEBUGGING MODE is not set in the ENVIRONMENT DIVISION of the source program all debugging lines are treated as comments.

$L -- list the input code on the screen as the program

is compiled. Default is on. Error messages are displayed at the terminal in any case.

$P -- Productions. List productions as they occur. Default is off.

$S -- sequence numbers are in the first six positions of each record. Default is off.

$T -- Tokens. List tokens from the scanner. Default is off.

$W -- Create a list file. Default is off. A listing file is created when this toggle is set. When this toggle is not set the "LST" file will only contain error messages.

IV. RUN TIME CONVENTIONS

This section explains how to run the compiler on the current system. The compiler expects to see a file with a type of CBL as the input file. In general, the input is free form. If the input includes sequence numbers then the compiler must be notified by setting the appropriate toggle. The compiler is started by typing COBOL <file-name>. Where the file name is the system name of the input file. There is no interaction required to start the second part of the compiler. The output file will have the same <file-name> as the input file, and will be given a file type of CIN. Any previous copies of the file will be erased. As with the CIN file a LST file will be created with the same file name as the input file and any previous LST files with that name will be erased.

The interpreter is started by typing EXEC <filename>. The first program is a loader, and it will display "NPS MICRO-COBOL LOADER VERS 1.Ø" followed by the display "LOAD FINISHED" to indicate successful completion. The run-time package will be brought in by the EXEC routine, and execution should continue without interruption. Succesful transfer of control to the interpreter will be indicated by the display "NPS MICRO-COBOL INTERPRETER VERS 1.Ø". Completion of program exection will be indicated by the display " X EXECTION ERROR(S)", where "X" is the number of

errors which occured during execution.

V. FILE INTERACTIONS WITH CP/M

The file structure that is expected by the program imposes some restrictions on the system. References 4 and 5 contain detailed information on the facilities of CP/M, and should be consulted for details. The information that has been included in this section is intended to explain where limitations exist and how the program interacts with the system.

All files in CP/M are on a random access device, and there is no way for the system to distinguish sequential files from files created in a random mode. This means that the various types of reads and writes are all valid to any file that has fixed length records. The restrictions of the ASSIGN statement prevent a file from being open for both random and sequential actions during one program.

Each logical record is terminated by a carriage return and a line feed. In the case of variable length records, this is the only end mark that exists. This convention was adopted to allow the various programs which are used in CP/M to work with the files. Files created by the editor, for example, will generally be variable length files. This convention removes the capability of reading variable length files in a random mode.

All of the physical records are 128 bytes in length, and the program supplies buffer space for these records in

addition to the logical records. Logical records may be of any desired length.

VI. ERROR MESSAGES

A. COMPILER FATAL MESSAGES

BR Bad read -- disk error, no corrective action can be taken in the program.

CL Close error -- unable to close the output file.

MA Make error -- could not create the output file.

MO Memory overflow -- the code and constants generated will not fit in the alloted memory space.

OP Open error -- can not open the input file, or no such file present.

SO Stack overflow -- the LALR(1) parsing stack has exceeded its maximum allowable size.

ST Symbol table overflow -- symbol table is too large for the allocated space.

WR Write error -- disk error, could not write a code record to the disk.

B. COMPILER WARNINGS

CC Carriage Control error -- The WRITE BEFORE/AFTER ADVANCING option can only be used with sequential files.

CE Close error -- attempted to close a non-existing file.

DD Duplicate Declaration -- the identifier name has been previously declared.

EL Extra levels -- only 10 levels are allowed.

FT File type -- the data element used in a read or write statement is not a file name.

IA Invalid access -- the specified options are not an allowable combination.

ID Identifier stack overflow -- more than 20 items in a GO -- DEPENDING statement.

IS Invalid subscript -- an item was subscripted but it was not defined by an OCCURS.

IT Invalid type -- the field types do not match for this statement.

LE Literal error -- a literal value was assigned to an item that is part of a group item previously assigned a value.

LV Literal value error -- the PICTURE clause field type does not match the VALUE clause literal type.

L7 Level 77 error -- level 77 used incorrectly.

MD Multiple decimals -- a numeric literal in a VALUE clause contains more than one decimal point.

MS Multiple signs -- a signed numeric literal in a VALUE clause contains more than one sign.

NF No file assigned -- there was no SELECT clause for this file.

NI Not implemented -- a production was used that is not

implemented.

NN Non-numeric -- an invalid character was found in a numeric string.

NP No production -- no production exists for the cuurrent parser configuration; error recovery will automatically occur.

NV Numeric value -- a numeric value was assigned to a non-numeric item.

OE Open error -- attempt to open a file that was not declared; or attempted to open a file for I-O that was not a RELATIVE file.

OL OCCURS LEVEL -- Ø1 and 77 levels can not contain an occurs clause.

PC Picture clause -- a pic clause exceeds 3Ø characters.

P1 More than one float symbol declared.

P2 Non-numeric data in repetition clause or missing right parenthesis.

P3 Invalid or incompatable symbol in pic clause.

P4 Invalid symbol(s) embedded within a float symbol only /,Ø,B,',' allowed.

P5 Invalid combination of symbols in pic clause, type cannot be determined.

P6 Number of possible numeric entries exceeds register length max is 18.

PF Paragraph first -- a section header was produced after a paragraph header, which is not in a section.

R1 Redefine nesting -- a redefinition was made for an item which is part of a redefined item.

R2 Redefine length -- the length of the redefinition item was greater than the item that it redefined. That is only allowed at the Ø1 level. This error message may be printed out one identifier past the redefining identifier record in which it occurred.

R3 Redefines misplaced -- a redefines was attempted in the FILE SECTION of the source program.

SE Scanner error -- the scanner was unable to read an identifier due to an invalid character.

SG Sign error -- either a sign was expected and not found, or a sign was present when not valid.

SL Significance loss -- the number assigned as a value is larger than the field defined.

TE Type error -- the type of a subscript index is not integer numeric.

UD Undeclared identifier -- the identifier was not declared.

UL Unresolved label -- label has not been referenced. This warning will be given to all references to external subroutines.

VE Value error -- a value statement was assigned to an item in the file section.

WL Wrong level error -- program attempted to write a record other than an Ø1 level record to an output

file.

C. INTERPRETER FATAL ERRORS

CL Close error -- the system was unable to close an output file.

CO Call stack Overflow -- insufficient memory available to transfer varable address' and/or return location for a subroutine call.

ME Make error -- the system was unable to make an output file on the disk.

NF No file -- an input file with the given name could not be opened.

OE Open Error -- attempt to open a file which was already open.

OP Open Error -- the system was unable to open a file.

PS Procedure Stack -- not enough memory to load all subroutines.

SO Subroutine Overflow -- subroutine symbol table overflow.

W1 Write non-sequential -- attempted to WRITE to a file opened for INPUT or a file opened for I-O when ACCESS was SEQUENTIAL.

W2 Wrong key -- attempted to change the key value to a lower value than the number of the last record written.

W3 Write input -- attempted to WRITE to a file opened for INPUT.

W4 Write non-empty -- attempted to WRITE to a non-empty record.

W5 Read output -- attempted to READ a file opened for OUTPUT.

W6 Rewrite error -- attempted to REWRITE to a file not opened for I-O.

W7 Rewrite error -- attempted to REWRITE a record before reading the file; or multiple REWRITE attempts without doing a READ between each.

D. INTERPRETER WARNING MESSAGES

EM End mark -- a record that was read did not have a carriage return or a line feed in the expected location.

GD Go to depending -- the value of the depending indicator was greater than the number of available branch addresses.

IC Invalid character -- an invalid character was loaded into an output field during an edited move. For example, a numeric character into an alphabetic-only field.

NE Numeric Error -- non-numeric data in an arithmetic operation.

W8 Write Error -- the system was unable to write to an output file on the disk. Disk may be full.

SI Sign Invalid -- the sign is not a "+" or a "-".

APPENDIX B

LIST OF MICRO-COBOL RESERVED WORDS

The following is a list of reserved words for MICRO-COBOL. The reserved words are the same as those specified for the HYPO-COBOL language, except where noted with an asterisk (*).

ACCEPT	END-IF *	MODE	ROUNDED
ACCESS	ENTER	MOVE	RUN
ADD	ENVIRONMENT	MULTIPLY	SAME
ADVANCING	EOF *	NEXT	SECTION
AFTER	EQUAL	NO *	SECURITY
ALPHABETIC	ERROR	NOT	SELECT
AND *	EXIT	NUMERIC	SENTENCE
ASSIGN	FD	OBJECT-COMPUTER	SEPARATE
AUTHOR	FILE	OCCURS	SEQUENTIAL
BEFORE	FILE-CONTROL	OF	SIGN
BLOCK	FILLER	OMITTED	SIZE
BY	FROM	OPEN	SOURCE-COMPUTER
CALL	GIVING *	OR *	SPACE
CLOSE	GO	ORGANIZATION	STANDARD
COBOL	GREATER	OUTPUT	STOP
COMP	I-O	PAGE	SUBTRACT
COMP-3 *	I-O-CONTROL	PERFORM	SYNC
COMPUTATIONAL*	IDENTIFICATION	PIC	THRU
COMPUTE *	IF	PROCEDURE	TIMES
CONFIGURATION	INDEXED *	PROGRAM	TO
DATA	INPUT	PROGRAM-ID	TRAILING
DATE-WRITTEN	INPUT-OUTPUT	QUOTE	UNTIL
DEBUGGING	INSTALLATION *	RANDOM	USAGE
DELETE	INVALID	READ	USING
DEPENDING	INTO *	RECORD	VALUE
DISPLAY	LABEL	RECORDS	VARYING *
DIVIDE	LEADING	REDEFINES	WITH *
DIVISION	LEFT	RELATIVE	WORKING-STORAGE
ELSE	LESS	REWRITE	WRITE
END	LINKAGE	RIGHT	ZERO

In addition the arithemetic operators "+", "-", "*", "/" and "**", and the comparison operators ">", "<" and "=" are in the reserved word list. None of these symbols are in in HYPØ COBOL but have been added to the grammar of NPS MICRO-COBOL

to enable greater flexiblity.

APPENDIX C

The MICRO-COBOL compiler and interpreter source files currently exist in the high level language PLM80 and are edited and compiled under the ISIS operating system on a INTEL Corporation MDS system. This is a description of the procedures required to compile and establish the programs to compile and interpret a MICRO-COBOL program. The MICRO-COBOL compiler/interpreter runs on any 8080 or Z-80 based microcomputer that operates under CP/M. The execution of the following four files will cause a MICRO-COBOL program to be compiled and executed:

1. COBOL.COM
2. PART2.COM
3. EXEC.COM
4. CINTERP.COM

These four files are created from the following six PLM80 source programs.

1. PART1.PLM
2. PART2.PLM
3. BUILD.PLM
4. READER.PLM
5. INTRDR.PLM
6. INTERP.PLM

The procedures used to create the four object files (COM files) involve compiling, linking, and locating each of the six source files under ISIS. The SID program is then used under CP/M to construct the executable files. Each of the following steps describe the action(s) to be taken and, where appropriate, the command string to be entered into the computer.

1. An ISIS system disk containing the PLM80 compiler is placed into drive A and a non-system disk containing the source programs is placed into drive B. It should be noted that drive A and B are the CP/M reference names for the drives while F1 and F2 are the ISIS reference names used for the associated disk drives.

2. Compile the PLM source program under ISIS using the the following command:

PLM80 :F1:<filename>.PLM DEBUG XREF

DEBUG saves the symbol table and line files for later use during debugging sessions. XREF causes a cross-reference listing, of all identifiers in the source program, to be created. The cross-reference listing includes each identifier and the associated line number where the identifier was declared and the line number of each occurence of the identifier in the source program [12].

3. Link the PLM80 object file.

LINK :F1:<filename>.OBJ, TRINT.OBJ, PLM80.LIB, TO
:F1:<filename>.MOD

See reference 11 for an explanation of PLM80.LIB. The TRINT.OBJ program interfaces the MON1 and MON2 functions of CP/M to the source program, allowing for the use of absolute addresses in referencing these functions.

4. Locate the object file.

LOCATE :F1:<filename>.MOD CODE(org address)

The "org address" is the address where the program will begin to be loaded into memory. The following are "org addresses" for the associated program:

PART1.MOD	103H
PART2.MOD	103H
INTERP.MOD	103H
INTRDR.MOD	80H
BUILD.MOD	103H
READER.MOD	0B000H

The "org addresses" above represent the ones used with a 62K byte CP/M system. The only address that would need to be changed if a different size system was used would be the one for IREADER.MOD. See appendix E for specifics on the address to use for IREADER.

4a. The two files INTRDR and IREADER just created by the

LOCATE command must be converted to "HEX FILES". By using the ISIS command OBJHEX <filename> the file will be converted to the "HEX file" <filename>.HEX.

5. Replace the ISIS system disk in drive A with a CP/M system disk and reboot the system.

6. Transfer the located ISIS file from the ISIS disk on drive B to the CP/M disk on drive A.

FROMISIS <filename>

6a. When transfering the "HEX files" to the CP/M disk use the following:

FROMISIS <filename>.HEX

7. Convert the ISIS file to a CP/M executable form.

OBJCPM <filename>

7a. The "HEX files" are not coverted to a CP/M format, but are left in HEX format.

7b. The file INTERP should be renamed to CINTERP using the command "REN CINTERP=INTERP" before the file is converted to CP/M executable form. This is nessacary because the ISIS operating system allows file names to be only six letters in length. When EXEC.COM is executed, the message "CINTERP.COM NOT FOUND" will be displayed if this step is not omitted.

At this point the object file is in machine readable

form and will run under CP/M when called properly. PART2.COM and CINTERP.COM are called by PART1.COM (COBOL.COM) and BUILD.COM (EXEC.COM), respectively and need no further work. COBOL.COM and EXEC.COM need to be constructed from the remaining four files.

COBOL.COM is created by entering the following commands:

1. SID PART1.COM
2. IREADER.HEX
3. R8600
4. A314A
5. JMP 0B000
6. Control-C
7. Save 56 COBOL.COM

See reference 7 for an explanation of the "I", "R", and "A" commands used above and ref 5 for an explanation of the "SAVE" command. Steps four and five above are used to patch the JUMP to READER referred to in the PART1.PLM program into the PART1.COM program. It should be noted that each time PART ONE is changed and recompiled the address of the "patch" instruction (step 4 above) will change. Use of the L command will aid in locating the address that needs to be changed. The assembly language code will have the following form: 314A JMP 314A.

EXEC.COM is created by entering the following commands:

1. SID BUILD.COM
2. IINTRDR.HEX
3. R1CØØ
4. CONTROL-C
5. SAVE 31 EXEC.COM

NPS MICRO-COBOL programs may now be executed in the following manner. The source program is named, <filename>.CBL. The command "COBOL <filename>", causes the MICRO-COBOL source program to be read into memory and compiled. During the compilation, the intermediate code file, <filename>.CIN, is written out to the disk as the code is generated. The command "EXEC <filename>", causes the file, <filename>.CIN, to be executed.

APPENDIX D

PART ONE AND PART TWO INTERNAL DATA STRUCTURES AND SIGNIFICANT VARIABLES

Within PART ONE and PART TWO, many significant data structures are used by the procedures which constitute the scanner and parser. Descriptions are given below for those structures regarded as important and necessary for future compiler development.

1. Interfacing Structures

ADD$END -- this variable is used to hold the end of file filler for the end of the source program.

BUFFER(11) -- byte array used to hold the filename and filetype if declared, of an input or output file in the SELECT CLAUSE of the FILE SECTION of a MICRO-COBOL source program.

BUFFER$END -- address variable which marks the last byte of the compiler input buffer which is a 128 byte buffer used for reading the source program.

ERROR$CTR(5) -- byte array used to hold a count of the total number of errors.

IN$ADDR -- address variable, default file control block used initially to hold the <filename.CBL> of the source program to be compiled.

IN$BUFF -- literal value, marks the first byte of the

compiler input buffer.

INPUT$FCB -- byte value, based at IN$ADDR(33), the base address of the default file control block of the source program.

LINE$CTR -- byte value that keeps track of the number of lines in the input file. Also used to write the line numbers to the list file.

LIST$BUFF(128) -- byte array, used as a 128 byte output buffer for loading the generated list file.

LIST$FCB(33) -- byte array for the list file, file control block.

LIST$PTR -- address value, used as an index into the list buffer (LIST$BUFF).

OUTPUT$BUFF(128) -- byte array, used as a 128 byte output buffer for loading the generated output (pseudo instructions) when writing to the intermediate code file.

OUTPUT$CHAR -- byte value, based at the OUTPUT$PTR; used to identify the particular byte of the output buffer (OUTPUT$BUFF) to which the next intermediate code instruction is to be written.

OUTPUT$END -- address variable, pointer to the end of the output buffer (OUTPUT$BUFF).

OUTPUT$FCB(33) -- byte array, the FCB for the intermediate code file <filename.CIN> established in PART ONE of the compiler and pasted to PART TWO of the compiler by IREADER module.

OUTPUT$PTR -- address value, used as an index into the output buffer (OUTPUT$BUFF).

POINTER -- address value, the address of the byte holding the next input character of the source program.

2. Debugging Structures

DEBUGGING -- logical byte value, toggle used in conjunction with ":" in a MICRO-COBOL source program text; allows for the compilation or non-compilation of the deugging statements following the ":".

ERROR -- logical byte value, toggle used to indicate an error condition and override a nolist condition thus allowing errors to be written to the list file reguardless of the write$lst toggle.

LIST$INPUT -- logical byte value, toggle used to display or not display a source program to the CRT during compilation.

NO$CODE -- logical byte value, toggle used to stop code generation for faster syntax checking.

PARMLIST(9) -- byte array used to hold the toggles set by the compiler developer or user upon execution of the command: COBOL <filename.CBL> $TOGGLES.

PRINT$PROD -- logical byte value, toggle used to print, in chronological order, at the CRT the production numbers of the compiler grammar rules used during a compilation of the source program.

PRINT$TOKEN -- logical byte value, toggle used to print tokens and the numbers assigned to them.

SEQ$NUM -- logical byte value, toggle used to indicate the presence of sequence numbers in the first six positions of each line of a source program being compiled.

WRITE$LST -- logical byte value, toggle used to indicate whether a list file is to be generated. A limited list file containing errors and the line being parsed at the time of the error(s) is always created.

UE$FLAG -- logical byte value, toggle used to indicate whether there is an undeclared varible.

3. Memory Structures

EOFFILLER -- literal value, used to test for the occurrence of an end of file character ("1AH" in CP/M), when reading the source program.

FREE$STORAGE -- first free address following PART ONE of the compiler; utilized as the base of the symbol table. This is the same value as HASHTABADDR in PART TWO of the compiler.

INITIAL$POS -- address value, the initial location of the IREADER module before it is copied to high memory at location MAX$MEMORY.

MAX$MEMORY -- address value, the location in high memory where the IREADER module is to be moved.

MAXINTMEM -- address value, the highest usable

addressable memory. This is the point where no more code can be generated due to insufficient memory.

NEXT$AVAILABLE -- address value, the pseudo machine memory address for the next machine instruction.

PART1$LEN -- the number of bytes of information saved in high memory after execution of PART ONE and used to initialize PART TWO module variables of the compiler.

PASS1$TOP -- this address is used in conjunction with PASS1$LEN for locating the fourty-eight bytes of information saved in PART ONE for use in PART TWO of the compiler.

RDR$LENGTH -- literal value representing the 255 bytes of the IREADER module to be moved from INITIAL$POS to MAX$MEMORY.

4. Scanner Structures:

ACCUM(51) -- an array of 51 bytes; the first byte contains a count of the total number of characters currently in the accumulator. This structure holds tokens as they are scanned, and will hold either a reserved word, a user defined identifier, or a literal.

COLLISION -- address varible, contained in first two bytes of an identifier's symbol table entry and indicates whether there is another identifier which hashes to the same hash table address. This address points to that identifier's address in the symbol table.

DISPLAY(88) -- an array of 74 bytes; the first byte

contains a count of the total number of characters (1-73) currently in the display buffer. Every line within a source program is loaded into this structure for subsequent printing to the CRT terminal during compilation.

EDIT$FLAG -- logical flag which denotes the fact that a '$' symbol has been loaded into the DISPLAY array during compilation. When set the characters within DISPLAY will be printed one at a time, until the entire line is printed.

HASH$TABLE$ADDR -- the base of the symbol table generated in PART ONE, used as the base of the hashtable.

HASHTABADDR -- this was the address of the bottom of the symbol table generated in PART ONE of the compiler, and saved for Part two.

INPUT$STR -- literal value (32), returned to the LALR(1) parser anytime the token contained in the ACCUM is not a reserved word or literal.

LITERAL -- literal value (15), returned to the LALR(1) parser anytime the first character encountered by the scanner is a quote ('), prior to loading the ACCUM.

MAX$LEN -- length of the longest reserved word allowed by MICRO-COBOL.

5. Parser Structures:

BUFFER(31) -- byte array used to store edited PICTURE CLAUSE characters for subsequent intermediated code generation.

COMPILING -- logical byte value which indicates that compiling is taking place or not in PART ONE or PART TWO; set to FALSE whenever the statestack of the LALR(1) parser is reduced to a recognizable finished state.

CUR$SYM -- address variable that holds the address of the current symbol being accessed in the symbol table.

DUP$IDEN$ARRAY(24) -- address array that holds the symbol address for all files declared in the INPUT-OUTPUT SECTION of a source program. When the FILE SECTION entry for the file is encountered the array is searched to determine if the file was declared and to insure that a FILE SECTION entry had not been previously made.

FILE$DESC$FLAG -- logical byte value; indicates whether the compiler is compiling the FILE DESCRIPTION SECTION of a source program or not.

FILESECEND --logical byte value set whenever the parser has parsed passed the FILE SECTION of a source program.

HOLD$LIT(51) -- byte array, first byte contains a count of the total number of characters currently stored in the HOLDLIT buffer which is used to hold characters for a VALUE CLAUSE.

ID$STACK(10) -- address array which functions as a stack and is used to hold the addresses of identifiers at both the record and elementary levels. Whenever a record identifier has nested elementary field identifiers it is saved on the

ID$STACK. Also, anytime a record identifier has succeeding record identifiers redefining it, it is saved on the ID$STACK. In the case of multiple record descriptions in a file description of the FILE SECTION, the record descriptions following the first record are assumed redefinitions.

ID$STACK$PTR -- a byte index variable into the ID$STACK array.

MAXIDLEN -- a numeric value (12), maximum length of any user defined identifier.

MP -- byte index variable into the VALUE array.

MPP1 -- byte index variable into the VALUE array, one byte above MP index.

NEXT$SYM -- this address indicates the next available free space for a symbol table entry.

PENDING$LITERAL -- byte value (Ø,1,2,3,4,5), indicates the category of the target input to a VALUE CLAUSE.

PENDINGLITID -- byte value (Ø,1,2,3,4,5), which is saved to indicate the category of the most recently encountered target input to a VALUE CLAUSE.

PRODUCTION -- byte value, determined by the parser and indicates the next semantic action to be taken by the compiler.

REDFF -- logical byte value which allows the testing of an identifier's storage value size against the storage value size of a second identifier that redefines the first. Set to

TRUE when there are multiple record descriptions within a FD BLOCK in the FILE SECTION, or when a record or elementary identifier declaration in the WORKING STORAGE SECTION contains a REDEFINES CLAUSE.

REDEF$FLAG -- logical byte value, used to denote the scanning and parsing of the FILE SECTION of a source program, helps in identifying duplicate identifiers within this section.

REDEF$ONE -- address variable that holds the symbol table address of the identifier being redefined by another identifier.

REDEF$TWO -- an address variable that contains the symbol table address of an identifier which redefines another identifier.

SP -- a byte index for the STATESTACK array and the VALUE array; points to the top of the STATESTACK array.

STATE -- a byte value numeric quantity that indicates the current parser state.

STATESTACK(40) -- a byte array which stacks the states (production sequences) the parser passes through while compiling a source program.

TRUNC$FLAG -- logical byte value that indicates numeric truncation of an identifier's VALUE CLAUSE input hasn't occurred, because the identifier's associated PICTURE CLAUSE has not been scanned and parsed.

VALUE(40) -- an address array that holds addresses of

identifiers, specific attributes of these identifiers and attributes of the current source program statement or sentence being parsed.

VARC(51) -- a byte array, the first byte holds the count of the total number of characters within it, used to hold all the ASCII characters of tokens scanned within the source program, excluding reserved words; for subsequent analysis and processing.

VALUE$FLAG -- a logical byte that is set anytime an identifier has an associated VALUE CLAUSE; used primarily to recognize the occurrence of a PICTURE CLAUSE before the VALUE CLAUSE or when a record entry has a VALUE CLAUSE, but no associated PICTURE CLAUSE except for those in its elementary field identifiers.

VALUE$LEVEL -- a byte value which saves the level number of a record identifier which doesn't have an associated PICTURE CLAUSE.

APPENDIX E

MACHINE DEPENDENT VARIABLES

The NPS MICRO-COBOL compiler/interpreter is designed to operate on any 8080 or Z80 based microcomputer operating under CP/M with at least 20K bytes of memory. The PLM80 source files have been written in such a way, that certain variables must be altered in the source code to take advantage of the machine that the programs are going to be operating on. This appendix covers those programs and the variables that must be altered.

1. PART1.PLM

This program has two variables that are memory size dependent, MAX$MEMORY and MAX$INT$MEMORY. The variable MAX$MEMORY is set to 100H bytes below the base of the BDOS and is used for the beginning address of the IREADER routine. The variable MAXINTMEMORY is set to the base address of the BDOS and is used as the upper limit for the intermediate code file.

2. PART2.PLM

This program also has two variables that are memory size dependent, MAX$MEMORY and PASS1$TOP. In this program MAX$MEMORY is set to the base address of the BDOS while PASS1$TOP is set to 100H bytes below the base of the BDOS.

3. READER.PLM

Although, this program does not have any memory size

dependent variables the program must be modified to execute properly. When using the LOCATE command, under ISIS, this routine must be located 100H bytes below the BDOS of the system. This address would correspond to the values of MAX$MEMORY in PART2.PLM and MAXINTMEMORY in PART1.PLM.

4. BUILD.PLM

This program has one memory size dependent variable, INTERP$ADDRESS must be set to the same address as CODE$START in INTERP.PLM.

5. INTERP.PLM and INTRDR.PLM

These two programs have no variables that need to be altered.

6. GENERAL INFORMATION

The current version of the NPS MICRO-COBOL compiler/interpreter is designed for continued development and certain variables are not set to make optimal use of memory. The variable NEXT$AVAILABLE, in PART1.PLM, is set to 3502H and CODE$START, in INTERP.PLM, is set to 3500H. Normally, CODE$START would be set to the address immediately following the last address in CINTERP.COM and NEXT$AVAILABLE would be set two bytes above that address. These address are currently set approximately 450H bytes above where they should be located, to allow for testing and expansion of the interpreter. As soon as implementation is completed these two addresses can be reset to appropriate values.

APPENDIX F

MICRO-COBOL PARSE TABLE GENERATION

The parse tables for NPS Micro-Cobol were generated on the IBM 360 using the LALR(1) parse table generater described in reference 20. There are basically two steps involved in generating the tables. First, a deck of cards containing the grammar is entered into the computer using the following JCL:

```
//PROGNAME JOB (2320,0417,CS91),'optional data',TIME=5
//GO EXEC PGM= LALR,REGION=220K
//STEPLIB DD DSN=F0119.LALR,UNIT=2314,
      VOL=SER=LINDA,DISP=SHR
//SYSPRINT DD SYSOUT=A,DCB=(RECFM=FB,
      LRECL=133,BLKSIZE=3325),
// SPACE=(CYL,(1,1))
//NONTERM DD SPACE=(CYL,(1,1)),UNIT=SYSDA
//FSMDATA DD SPACE=(CYL,(1,1)),UNIT=SYSDA
* //PTABLES DD SYSOUT=B,
      DCB=(RECFM=FB,LRECL=80,BLKSIZE=800)
//SYSIN DD *
```

* This card can be replaced by //PTABLES DD SYSOUT=DUMMY to surpress the card punching feature. This allows modifications to be made without wasting cards until a new LALR(1) grammer is produced.

The ouput from this run is a listing and a card deck containing the tables in XPL compatable format. This deck is then translated into PLM compatible format using the following JCL and an XPL program which is available in the card deck library in the Computer Science Department at the Naval Postgraduate School.

```
//EXEC XCOM
//COMP.SYSIN DD *
//GO.SYSPUNCH DD SYSOUT=B,
      DCB=(RECFM=FB,LRECL=80,BLKSIZE=800)
//GO.SYSIN DD *
```

The tables are then transferred to a diskette and edited into the PLM80 source program using the ISIS COPY and EDIT features on the INTEL MDS System. See APPENDIX H for the procedures to transfer files from the IBM-360 to a floppy diskette.

APPENDIX G

LIST OF INOPERATIVE CONSTRUCTS

The following is a list of MICRO-COBOL elements that either have not been implemented.

CLOSE - multiple closes

OPEN - multiple open's

The following HYPO-COBOL elements are part of NPS MICRO-COBOL only to the extent that they are defined in the grammar. No code has been written to support them.

COMPUTE

AND and OR

ENTER

COMP and COMPUTATIONAL (binary arithmetic storage and operations)

INDEXED

MULTI-DIMENSION tables

APPENDIX H

IBM TO MICROCOMPUTER TRANSFER PROCEDURES

A CP/M operating system program was written by Prof. Kodres for the express purpose of transferring ASCII files from the IBM CP/CMS system. In order to use this program, several equipment requirements must be met: a.) Reserve the appropriate Intel MDS system in the Microcomputer Lab. b.) Call 646-2721(computer-center) to reserve a high speed(1200 baud) line to the micro-lab. c.) Connect the line marked "IBM 1200 BAUD" line to the "black box" marked IBM, which contains line drivers for the RS-232 circuit. Check that the toggle switch is in the up/raised position. d.) Connect the serial connector coming off the MODIFIED single board computer (marked with a yellow dot) to the other end of the line driver box. All of the other boards in the MDS are unmodified with the exception of times when hardware experimentation is being conducted by various groups of students and/or facalty.

To commence communication with the 360 - invoke the CP/M program IBM.COM - an executible file. The program is loaded and executed by typing "IBM filename.filetype", where "filename.filetype" is selected by the user as the CP/M file which will be created as a result of a file transfer. Successful completion of the above steps will result in the following data being displayed on the CRT:

(crt echo? y/n)	Answer "y"
(n)	Placed by the CP/M program
	Enter a <CR>
caCP-67 Online	Normal CP/CMS signon message

At this point login to CP/CMS in a normal manner. Files are transfered using the CMS command "PRINT" followed by the name of the file to be transfered followed by a control-R. This will cause the MDS to be put into the receive mode. A <CR> will start the file transfer. The CRT should display the following for a successful file transfer.

PRINT cmsfilename cmsfiletype	Enter a control-R
(R)	Puts MDS in receive mode
(R. CREATED filename.filetype)	Enter a <CR>
(---- bytes received END R)	Enter a <CR> to re-enter CP
	Enter a control-C to reboot

Each file transfer must be done with a separate invocation of the IBM file as all files will be transfered to the file named when IBM is invoked. Before rebooting for the last time logout of CP/CMS in the normal manner and call 2721 and inform the computer center that the high speed line is available for other user's.

APPENDIX I

DEBUGGING NPS MICRO-COBOL USING SID

Note: Steps two and three are optional. They are used if the line numbers in the program listing are to be used as well as the symbols for pass points.

PART ONE.

1. SID COBOL.COM PART1.SYM
2. I* PART1.LIN
3. R <ret>
4. I<file name.CBL> $<compilier toggles as required>
5. Set desired passpoints

PART TWO.

1. SID COBOL.COM PART2.SYM
2. I* PART2.LIN
3. R<ret>
4. i<file name.CBL> $<compilier toggles as required>
5. T50
6. G,0B000
7. T50
8. G,100
9. Set desired passpoints

INTERPRETER. Note: Use only SYM or LIN files but not both.

1. SID EXEC.COM CINTERP.SYM
2. I* CINTERP.LIN
3. R<ret>
4. I<file name.CIN>
5. G,22E
6. T25
7. G,100
8. Set desired passpoints

These instructions are designed to get the programs to the proper place to be able to use SID. See reference [8] for instructions on how to use SID commands. It should be noted that changes to the routine BUILD will change instruction 5 in the INTERPRETER command list. That command is intended to stop after BUILD has finished executing and is the location of the last instruction in that module.

```
$ TITLE('NPS MICRO-COBOL COMPILER PART1') PAGEWIDTH(80)
  PAGELENGTH(60)
PART1:DO;

              /*    COBOL COMPILER - PART 1    */

              /*    NORMALLY LOCATED AT 103H    */

          /*    GLOBAL DECLARATIONS AND LITERALS    */

DECLARE DCL LITERALLY 'DECLARE',
        LIT LITERALLY 'LITERALLY';

DCL CR                LIT       '13',
    EOFFILLER         LIT       '1AH', /* END OF RECORD FILLER */
    FALSE             LIT       '0',
    ERROR             BYTE      INITIAL(FALSE),
    FILE$DESC$FLAG    BYTE      INITIAL(FALSE),
    UI$FLAG           BYTE,     /*UNDECLARED VAR FLAG*/
    FOREVER           LIT       'WHILE TRUE',
    INITIAL$POS       ADDRESS   INITIAL(3600H),
    LF                LIT       '10',
    MAX$MEMORY        ADDRESS   INITIAL(0B000H),
    QUOTE             LIT       '27H',
    PARMLIST(9)       BYTE      INITIAL('         '),
    PARMS             LIT       '6DH',
    PASS1$LEN         ADDRESS   INITIAL(353),
    POUND             LIT       '23H',
    PROC              LIT       'PROCEDURE',
    RDR$LENGTH        LIT       '255',
    TRUE              LIT       '1';

DCL     MAXLNO LITERALLY '138', /* MAX LOOK COUNT */
        MAXPNO LITERALLY '156', /* MAX PUSH COUNT */
        MAXRNO LITERALLY '110', /* MAX READ COUNT */
        MAXSNO LITERALLY '253', /* MAX STATE COUNT */
        STARTS LITERALLY '1',   /* START STATE */
        PRODNO LITERALLY '97',  /* NUMBER OF PRODUCTIONS */
        PROCC  LITERALLY '48',   /* PROCEDURE */
        TERMNO LITERALLY '64';  /* TERMINAL COUNT */
DCL READ1 (*) BYTE
    DATA(0,61,50,60,33,8,25,63,2,33,55,62,11,33,33,41,40,36
    ,46,9,19,39,6,26,34,59,3,14,15,18,20,33,29,51,33,1,44,40
    ,38,45,1,1,1,1,1,1,1,1,1,10,1,41,1,1,1,40,1,35,42,51,40
    ,41,1,1,40,16,17,22,30,23,24,58,54,57,43,37,48,1,7,52,1
    ,33,1,33,33,47,1,33,1,33,1,33,1,33,49,27,33,39,4,35,56
    ,42,1,1,33,5,12,13,21,22,28,1,64,1,23,24,58,31,53);
```

```
DCL LOOK1(*) BYTE
    DATA(0,8,0,25,0,9,19,0,44,0,44,0,1,0,54,0,57,0,43,0,37,0
    ,52,0,1,0,49,0,4,0,35,0,56,0,42,0,1,0,2,0,33,0,1,0,1,0,11
    ,0,64,0,7,0,33,0,33,0,33,0);
DCL APPLY1(*) BYTE
    DATA(0,0,0,0,0,0,0,0,9,10,12,14,16,20,0,0,0,0,0,0,107,0,0
    ,106,0,0,0,0,0,0,103,0,28,0,0,0,0,98,0,0,0,96,0,0,0,0,13
    ,18,0,108,109,110,0,0,0,0,0,101,0,0,56,0,0,24,31,39,40,0
    ,22,41,42,54,58,90,99,100,0);
DCL READ2 (*) BYTE
    DATA(0,68,59,67,168,27,38,70,22,252,63,69,28,253,231,53
    ,47,114,115,242,243,45,232,233,235,234,23,249,248,251,250
    ,247,189,188,184,9,245,49,212,211,7,8,11,13,15,2,3,111,16
    ,173,4,52,21,14,19,50,12,187,186,185,46,51,20,10,48,31,32
    ,34,40,36,37,66,62,65,55,44,157,17,26,60,112,169,160,169
    ,169,57,162,169,164,169,166,169,172,169,58,224,253,209
    ,24,42,64,54,222,196,253,25,29,113,33,35,39,18,71,179,36
    ,37,66,41,61);
DCL LOOK2(*) BYTE
    DATA(0,5,139,6,140,30,30,141,43,142,56,143,144,72,74,145
    ,75,146,76,147,77,148,81,149,150,84,89,151,92,214,93,239
    ,94,152,95,153,205,96,98,200,99,213,227,101,154,102,103
    ,192,105,155,156,107,108,216,109,218,110,204);
DCL APPLY2(*) BYTE
    DATA(0,0,121,158,118,117,119,159,83,122,85,86,87,88,82,80
    ,126,79,170,135,178,177,106,181,180,182,127,183,175,133
    ,195,194,100,130,78,134,129,203,202,104,128,208,207,210
    ,120,199,137,138,136,221,221,221,220,123,132,97,131,230
    ,229,240,237,236,241,215,91,125,124,90,116,73,238,190,225
    ,223,198,198,197);
DCL INDEX1(*) BYTE
    DATA(0,1,2,3,4,5,6,7,8,4,4,9,4,9,4,10,4,11,9,117,4,12,13
    ,13,9,14,15,16,13,17,19,9,21,22,26,27,32,34,35,9,9,13,13
    ,36,37,38,40,41,42,43,44,45,46,47,13,48,36,49,13,50,51,52
    ,53,54,55,56,57,60,61,62,63,64,65,69,72,73,74,75,76,77,78
    ,79,80,82,84,86,88,90,92,94,95,97,98,99,100,101,65,102,8
    ,13,103,105,105,12,111,112,113,117,9,9,9,1,3,5,8,10,12
    ,14,16,18,20,22,24,26,28,30,32,34,36,38,40,42,44,46,48,50
    ,52,54,56,201,161,244,246,246,206,165,163,167,219,171,174
    ,226,176,191,228,217,193,1,2,3,4,4,5,5,6,6,7,7,8,8,15,15
    ,16,17,17,18,18,19,19,20,22,22,23,23,23,25,25,25,26,26,27
    ,27,28,28,29,29,30,32,32,34,35,35,36,36,37,39,39,40,40,41
    ,41,41,41,41,43,43,44,44,45,45,46,46,49,53,53,54,54,55,55
    ,56,56,57,57,57,57,57,57,57,57,57,57,57,59,59,59,60,60,62,
    ,62,62,62,62,63,68);
DCL INDEX2(*) BYTE
    DATA(0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1
    ,1,1,1,2,2,1,1,4,1,5,2,1,1,1,1,1,1,1,1,2,1,1,1,1,1,1,1,1
    ,1,1,1,1,1,1,1,1,1,1,1,1,3,1,1,1,1,1,1,4,3,1,1,1,1,1,1,1,1
    ,2,2,2,2,2,2,2,1,2,1,1,1,1,1,4,1,1,1,2,6,6,1,1,1,4,2,1,1
    ,1,2,2,3,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2
```

```
    ,2,5,6,30,43,56,72,74,75,76,77,81,84,89,94,95,102,105,107
    ,3,7,3,3,0,3,0,3,0,3,0,0,1,7,0,8,1,0,6,0,0,1,3,0,1,1,2,1
    ,0,0,0,0,0,1,0,2,0,0,1,2,0,1,5,3,0,0,1,4,0,0,0,1,2,1,2,2
    ,2,0,2,3,0,3,0,0,1,4,0,0,1,0,0,0,0,1,1,1,1,1,1,2,2,3,1,1
    ,1,0,0,0,0,0,0,0,0,0,0,0,0,0);
    /* JOINT DECLARATIONS
    THESE ITEMS ARE DECLARED TOGETHER IN THIS SECTION
    IN ORDER TO FACILITATE THEIR BEING SAVED FOR
    THE SECOND PART OF THE COMPILER. */

DCL DEBUGGING          BYTE      INITIAL(FALSE),
    ERROR$CTR(5)       BYTE      INITIAL('    0'),
    LINE$CTR(5)        BYTE,
    LIST$BUFF(128)     BYTE,
    LIST$FCB(33)       BYTE      INITIAL(0,'        LST',0,0,0
        ,0),
    LIST$INPUT         BYTE      INITIAL(TRUE),
    LIST$PTR           ADDRESS,
    MAX$INT$MEM        ADDRESS   INITIAL(0B100),
    NEXT$AVAILABLE     ADDRESS   INITIAL(3502H),
    NEXT$SYM           ADDRESS,
    NO$CODE            BYTE      INITIAL(FALSE),
    OUTPUT$BUFF(128)   BYTE,
    OUTPUT$FCB(33)     BYTE      INITIAL(0,'        CIN',0,0,0
        ,0),
    OUTPUT$PTR         ADDRESS,
    POINTER            ADDRESS   INITIAL(100H),
    PRINT$PROD         BYTE      INITIAL(FALSE),
    PRINT$TOKEN        BYTE      INITIAL(FALSE),
    SEQ$NUM            BYTE      INITIAL(FALSE),
    WRITE$LST          BYTE      INITIAL(FALSE),
    FREE$STORAGE       ADDRESS   INITIAL(3800H),
    FILE$SEC$END       BYTE      INITIAL(FALSE),

    /*  I O BUFFERS AND GLOBALS */
    IN$ADDR            ADDRESS INITIAL(5CH),
    INPUT$FCB          BASED IN$ADDR(33) BYTE,
    LIST$CHAR          BASED LIST$PTR BYTE,
    LIST$END           ADDRESS,
    OUTPUT$CHAR        BASED OUTPUT$PTR BYTE,
    OUTPUT$END         ADDRESS;

MON1: PROC (F,A) EXTERNAL;
    DCL A ADDRESS, F BYTE;
END MON1;

MON2: PROC (F,A) BYTE EXTERNAL;
    DCL F BYTE, A ADDRESS;
END MON2;

BOOT: PROC EXTERNAL;
```

```
    END BOOT;

PRINT$CHAR: PROC (CHAR);
    DCL CHAR BYTE;
    CALL MON1 (2,CHAR);
END PRINTCHAR;

WRITE$OUTPUT: PROC(BUFF,FCB); /* WRITES OUT A BUFFER */
    DCL (BUFF,FCB) ADDRESS;
    CALL MON1(26,BUFF);     /* SET DMA */
    IF MON2(21,FCB) <> 0 THEN
        DO;
            CALL MON1(9,.('WR$'));
            CALL BOOT;
        END;
    CALL MON1(26,80H);     /* RESET DMA */
END WRITE$OUTPUT;

WRITE$TO$DISK: PROC(CHAR);
    DCL CHAR BYTE;
    IF (LIST$PTR := LIST$PTR + 1) > LIST$END THEN
        DO;
            CALL WRITE$OUTPUT(.LIST$BUFF,.LIST$FCB);
            LIST$PTR = .LIST$BUFF;
        END;
    LIST$CHAR = CHAR;
END WRITE$TO$DISK;

PRINT: PROC (A);
    DCL (A,ADDR) ADDRESS,CHAR BASED ADDR BYTE;
    ADDR = A;
    DO WHILE CHAR <> '$';
        CALL WRITE$TO$DISK(CHAR);
        ADDR = ADDR + 1;
    END;
    CALL MON1 (9,A);
END PRINT;

CRLF:PROC;
    CALL MON1(9,.(CR,LF,'$'));
END CRLF;

DCRLF: PROC;
    CALL WRITE$TO$DISK(CR);
    CALL WRITE$TO$DISK(LF);
END DCRLF;

INC$CTR: PROC(BASE);
    DCL BASE ADDRESS, CTR BYTE, B$BYTE BASED BASE (1) BYTE,
        TEN LIT '3AH';
    CTR = 4;
```

```
    DO WHILE (B$BYTE(CTR) := B$BYTE(CTR) + 1) = TEN;
        B$BYTE(CTR) = '0';
        IF CTR > 0 THEN
            IF B$BYTE(CTR := CTR - 1) = ' ' THEN
                B$BYTE(CTR) = '0';
    END;
END INC$CTR;

PRINT$ERROR: PROC (CODE);
    DCL I BYTE,CODE ADDRESS,CODE1(6) ADDRESS;
    IF CODE = FALSE THEN
        DO;
            DO I = 0 TO 5;
                CODE1(I) = 0;
            END;
            I = 0;
        END;
    ELSE IF CODE = TRUE THEN
        DO;
            I = 0;
                DO WHILE((I <> 6) AND ( CODE1(I) <> 0));
                    CALL PRINTCHAR(HIGH(CODE1(I)));
                    CALL PRINTCHAR(LOW (CODE1(I)));
                    CALL WRITE$TO$DISK(HIGH(CODE1(I)));
                    CALL WRITE$TO$DISK(LOW (CODE1(I)));
                    CALL CRLF;
                    CALL DCRLF;
                    CODE1(I) = 0;
                    I = I + 1;
                END;
            I = 0;
            ERROR = FALSE;
        END;
    ELSE IF (CODE = 'NP') OR (CODE = 'SL')
                          OR (CODE = 'NV') THEN
        DO;
            ERROR = TRUE;
            CALL PRINTCHAR(HIGH(CODE));
            CALL PRINTCHAR(LOW(CODE));
            CALL INC$CTR(.ERROR$CTR(0));
            IF CODE <> 'NP' THEN
                DO;
                    CALL CRLF;
                    CALL DCRLF;
                END;
        END;
    ELSE
        DO;
            ERROR = TRUE;
            IF I <> 6 THEN
                DO;
```

```
                    CODE1(I) = CODE;
                    I = I + 1;
                END;
            CALL INC$CTR(.ERROR$CTR(0));
        END;
END PRINT$ERROR;

FATAL$ERROR: PROC(REASON);
    DCL REASON ADDRESS;
    CALL PRINT$ERROR(REASON);
    CALL PRINT$ERROR(TRUE);
    CALL BOOT;
END FATAL$ERROR;

OPEN: PROC;
    IF MON2 (15,IN$ADDR) = 255 THEN CALL FATAL$ERROR('OP');
END OPEN;

MORE$INPUT: PROC BYTE;
    DCL DCNT BYTE;
    IF (DCNT := MON2(20,.INPUT$FCB)) > 1 THEN
        CALL FATAL$ERROR('BR');
    RETURN NOT(DCNT);
END MORE$INPUT;

MAKE: PROC(FCB);
   DCL FCB ADDRESS;
    /* DELETES ANY EXISTING COPY OF THE OUTPUT FILE
         AND CREATES A NEW COPY*/
    CALL MON1(19,FCB);
    IF MON2(22,FCB) = 255 THEN CALL FATAL$ERROR('MA');
END MAKE;

MOVE: PROC(SOURCE, DESTINATION, COUNT);
    DCL (SOURCE,DESTINATION,COUNT) ADDRESS,
    (S$BYTE BASED SOURCE, D$BYTE BASED DESTINATION) BYTE;
    DO WHILE (COUNT := COUNT - 1) <> 0FFFFH;
        D$BYTE = S$BYTE;
        SOURCE = SOURCE + 1;
        DESTINATION = DESTINATION + 1;
    END;
END MOVE;

FILL: PROC(ADDR,CHAR,COUNT);
    DCL (ADDR,COUNT) ADDRESS,
    (CHAR,DEST BASED ADDR) BYTE;
    DO WHILE (COUNT := COUNT - 1) <> 0FFFFH;
         DEST = CHAR;
         ADDR = ADDR + 1;
    END;
END FILL;
```

```
    /*   *  *  *  *  *  SCANNER LITS *  *  *  *  */
DCL INPUT$STR       LIT         '33',
    LITERAL         LIT         '15',
    PERIOD          LIT         '1';

    /*  * * * * SCANNER TABLES * * * *    */
DCL TOKEN$TABLE (*) BYTE DATA
    /* CONTAINS THE TOKEN NUMBER ONE LESS THAN THE FIRST
    RESERVED WORD FOR EACH LENGTH OF WORD  */
    (0,0,1,4,5,15,22,33,40,46,49,51,53,58,60,61).

TABLE(*) BYTE DATA('FD','OF','TO','PIC','COMP','DATA','FILE'
    ,'LEFT','MODE','SAME','SIGN','SYNC','ZERO','BLOCK'
    ,'LABEL','QUOTE','RIGHT','SPACE','USAGE','VALUE','ACCESS'
    ,'ASSIGN','AUTHOR','COMP-3','FILLER','OCCURS','RANDOM'
    ,'RECORD','SELECT','DISPLAY','INDEXED','LEADING'
    ,'LINKAGE','OMITTED','RECORDS','SECTION','DIVISION'
    ,'RELATIVE','SECURITY','SEPARATE','STANDARD','TRAILING'
    ,'DEBUGGING','PROCEDURE','REDEFINES','PROGRAM-ID'
    ,'SEQUENTIAL','ENVIRONMENT','I-O-CONTROL','DATE-WRITTEN'
    ,'FILE-CONTROL','INPUT-OUTPUT','INSTALLATION'
    ,'ORGANIZATION','COMPUTATIONAL','CONFIGURATION'
    ,'IDENTIFICATION','OBJECT-COMPUTER','SOURCE-COMPUTER'
    ,'WORKING-STORAGE'),

OFFSET (16) ADDRESS
    /*  NUMBER OF BYTES TO INDEX INTO THE TABLE FOR EACH
        LENGTH */
    INITIAL (0,0,0,6,9,45,80,134,183,231,258,278,
         300,360,386,400),

WORD$COUNT (*) BYTE DATA
    /* NUMBER OF WORDS OF EACH SIZE */
    (0,0,3,1,9,7,9,7,6,3,2,2,5,2,1,3),

    ACCUM$LEN$P$1  LIT        '51', /* ACCUM$LENG PLUS 1 */
    ACCUM (ACCUM$LEN$P$1) BYTE,
    ACCUM$LENG     LIT        '50',
    ADD$END(*)     BYTE       DATA('      PROCEDURE'),
    BUFFER$END     ADDRESS    INITIAL(100H),
    CHAR           BYTE       INITIAL(CR),
    DISPLAY(88)    BYTE       INITIAL(5,'  1 '),
    FIRST$LINE     BYTE       INITIAL(TRUE),
    FORM$FEED      LIT        '0CH',
    HOLD           BYTE,
    INBUFF         LIT        '80H',
    LOOKED         BYTE       INITIAL(FALSE),
    MAX$LEN        LIT        '15',
    NEXT           BASED      POINTER  BYTE,
    TAB            LIT        '09',
```

```
    TOKEN           BYTE;     /*RETURNED FROM SCANNER */

    /* * * * *   PROCEDURES USED BY THE SCANNER * * * * */

NEXT$CHAR: PROC BYTE;
    IF LOOKED THEN
        DO;
            LOOKED = FALSE;
            RETURN (CHAR := HOLD);
        END;
    IF (POINTER:=POINTER + 1) >= BUFFER$END THEN
        DO;
            IF NOT MORE$INPUT THEN
                DO;
                    BUFFER$END = .MEMORY;
                    POINTER = .ADD$END;
                END;
            ELSE POINTER = INBUFF;
        END;
    IF NEXT = EOFFILLER THEN
        DO;
            BUFFER$END = .MEMORY;
            POINTER = .ADD$END;
        END;
    RETURN (CHAR := NEXT);
END NEXT$CHAR;

GET$CHAR: PROC;
    CHAR=NEXT$CHAR;
END GET$CHAR;

DISPLAY$LINE: PROC;
    DCL  I BYTE;
    DO I = 1 TO DISPLAY(0);
        IF LIST$INPUT OR ERROR THEN CALL
            PRINTCHAR(DISPLAY(I));
        IF WRITE$LST OR ERROR THEN
            CALL WRITE$TO$DISK(DISPLAY(I));
    END;
    CALL INC$CTR(.DISPLAY(0));
    DISPLAY(0) = 5;
END DISPLAY$LINE;

LOAD$DISPLAY: PROC;
    IF DISPLAY(0) < 87 THEN
        DISPLAY(DISPLAY(0) := DISPLAY(0) + 1) = CHAR;
    CALL GET$CHAR;
END LOAD$DISPLAY;

PUT: PROC;
    IF ACCUM(0) < ACCUM$LENG THEN
```

```
    ACCUM(ACCUM(0) := ACCUM(0) + 1) = CHAR;
    CALL LOAD$DISPLAY;
END PUT;

EAT$LINE: PROC;
    DO WHILE CHAR <> CR;
        CALL LOAD$DISPLAY;
    END;
END EAT$LINE;

GET$NO$BLANK: PROC;
    DCL I BYTE;
    DO FOREVER;
        IF (CHAR = ' ' OR CHAR = TAB) THEN CALL LOAD$DISPLAY;
        ELSE IF CHAR=CR THEN
            DO;
                IF FIRST$LINE THEN
                    DO;
                        FIRST$LINE = FALSE;
                        CALL GET$CHAR;
                    END;
                ELSE
                    DO;
                        CALL LOAD$DISPLAY;
                        CALL LOAD$DISPLAY;
                        CALL DISPLAY$LINE;
                        CALL PRINT$ERROR(TRUE);
                    END;
                DO WHILE CHAR = CR;
                    CALL LOAD$DISPLAY;
                    CALL LOAD$DISPLAY;
                    CALL DISPLAY$LINE;
                END;
                IF SEQ$NUM THEN
                    DO I = 1 TO 6;
                        CALL LOAD$DISPLAY;
                    END;
                IF CHAR = '*' THEN CALL EAT$LINE;
                ELSE IF CHAR = '/' THEN
                    DO;
                        IF LIST$INPUT THEN
                            CALL PRINT$CHAR(FORM$FEED);
                        IF WRITE$LST THEN
                            CALL WRITE$TO$DISK(FORM$FEED);
                        CALL EATLINE;
                    END;
                ELSE IF CHAR = ':' THEN
                    DO;
                        IF NOT DEBUGGING THEN CALL EAT$LINE;
                        ELSE CALL LOAD$DISPLAY;
                    END;
```

```
            END;
        ELSE RETURN;
    END;    /* END OF DO FOREVER */
END GET$NO$BLANK;

SPACE: PROC BYTE;
    RETURN (CHAR = ' ') OR (CHAR = CR) OR (CHAR = TAB);
END SPACE;

DELIMITER: PROC BYTE;
    IF CHAR <> '.' THEN RETURN FALSE;
    HOLD = NEXT$CHAR;
    LOOKED = TRUE;
    IF SPACE THEN
        DO;
            CHAR = '.';
            RETURN TRUE;
        END;
    CHAR = '.';
    RETURN FALSE;
END DELIMITER;

END$OF$TOKEN: PROC BYTE;
    RETURN SPACE OR  DELIMITER;
END END$OF$TOKEN;

GET$LITERAL: PROC BYTE;
    CALL LOAD$DISPLAY;
    DO FOREVER;
        IF CHAR = QUOTE THEN
            DO;
                CALL LOAD$DISPLAY;
                RETURN LITERAL;
            END;
        CALL PUT;
    END;
END GET$LITERAL;

LOOK$UP: PROC BYTE;
    DCL POINT ADDRESS,HERE BASED POINT(1) BYTE, I BYTE;

    MATCH: PROC BYTE;
        DCL J BYTE;
        DO J = 1 TO ACCUM(0);
            IF HERE(J - 1) <> ACCUM(J) THEN RETURN FALSE;
        END;
        RETURN TRUE;
    END MATCH;

    POINT = OFFSET(ACCUM(0)) + .TABLE;
    DO I = 1 TO WORD$COUNT(ACCUM(0));
```

```
        IF MATCH THEN RETURN I;
        POINT = POINT + ACCUM(0);
    END;
    RETURN FALSE;
END LOOK$UP;

RESERVED$WORD: PROC BYTE;
    DCL (NUMB,VALUE) BYTE;
    IF ACCUM(0) > MAX$LEN THEN RETURN 0;
    IF (NUMB := TOKEN$TABLE(ACCUM(0))) = 0 THEN RETURN 0;
    IF (VALUE := LOOK$UP) = 0 THEN RETURN 0;
    RETURN (NUMB + VALUE);
END RESERVED$WORD;

GET$TOKEN: PROC BYTE;
    ACCUM(0) = 0;
    CALL GET$NO$BLANK;
    IF CHAR = QUOTE THEN RETURN GET$LITERAL;
    IF DELIMITER THEN
        DO;
            CALL PUT;
            RETURN PERIOD;
        END;
    DO FOREVER;
        CALL PUT;
        IF END$OF$TOKEN THEN RETURN INPUT$STR;
    END; /* OF DO FOREVER */
END GET$TOKEN;

SCANNER: PROC;
    DCL CHECK BYTE;
    DO FOREVER;
        IF(TOKEN := GET$TOKEN) = INPUT$STR THEN
            IF (CHECK := RESERVED$WORD) <> 0 THEN
                TOKEN = CHECK;
        IF TOKEN <> 0 THEN RETURN;
        CALL PRINT$ERROR ('SE');
        DO WHILE NOT END$OF$TOKEN;
            CALL GET$CHAR;
        END;
    END;
END SCANNER;

PRINT$ACCUM: PROC;
    DCL I BYTE;
    DO I = 1 TO ACCUM(0);
        CALL PRINT$CHAR(ACCUM(I));
        CALL WRITE$TO$DISK(ACCUM(I));
    END;
    CALL CRLF;
    CALL DCRLF;
```

```
END PRINT$ACCUM;

PRINT$NUMBER: PROC(NUMB);
    DCL(NUMB,I,CNT,K) BYTE, J(*) BYTE DATA(100,10);
    DO I = 0 TO 1;
        CNT = 0;
        DO WHILE NUMB >= (K := J(I));
            NUMB=NUMB - K;
            CNT=CNT + 1;
        END;
        CALL PRINTCHAR('0' + CNT);
    END;
    CALL PRINTCHAR('0' + NUMB);
END PRINT$NUMBER;

INIT$SCANNER: PROC;
    DCL CON$CBL (*) BYTE DATA ('CBL'),(TESTFLAG,I) BYTE;
    CALL MOVE(PARMS,.PARMLIST,8);
    IF PARMLIST(0) = '$' THEN
    DO;
      I = 0;
      DO WHILE (TESTFLAG := PARMLIST(I := I + 1)) <> ' ';
        IF TESTFLAG = 'L' THEN LIST$INPUT  = NOT LIST$INPUT;
        IF TESTFLAG = 'S' THEN SEQ$NUM     = NOT SEQ$NUM;
        IF TESTFLAG = 'P' THEN PRINT$PROD  = NOT PRINT$PROD;
        IF TESTFLAG = 'T' THEN PRINT$TOKEN = NOT PRINT$TOKEN;
        IF TESTFLAG = 'C' THEN NO$CODE     = NOT NO$CODE;
        IF TESTFLAG = 'W' THEN WRITE$LST   = NOT WRITE$LST;
        IF TESTFLAG = 'D' THEN DEBUGGING   = NOT DEBUGGING;
      END;
    END;
    CALL MOVE(.CON$CBL,IN$ADDR + 9,3);
    CALL FILL(IN$ADDR + 12,0,5);
    CALL OPEN;
    IF NOT NO$CODE THEN
       DO;
         CALL MOVE(INADDR,.OUTPUT$FCB,9);
         OUTPUT$FCB(32) = 0;
         OUTPUT$END = (OUTPUT$PTR := .OUTPUT$BUFF - 1) + 128;
         CALL MAKE(.OUTPUT$FCB);
       END;
    CALL MOVE(INADDR,.LIST$FCB,9);
    LIST$FCB(32) = 0;
    LIST$END = (LIST$PTR := .LIST$BUFF - 1) + 128;
    CALL MAKE(.LIST$FCB);
    CALL GET$NO$BLANK; /* PRIME THE SCANNER */
    CALL PRINT$ERROR(FALSE);
    CALL PRINT(.('NPS MICRO-COBOL COMPILER VERSION 2.0',
        CR,LF,LF,'$'));
END INIT$SCANNER;
```

```
    /*  *  *  *  END OF SCANNER PROCEDURES  *  *  *  */

    /*  *  *  *  *  SYMBOL TABLE DECLARATIONS * * *  */

DCL
ADDR2                LIT                 '4',
CUR$SYM              ADDRESS,            /*SYMBOL BEING ACCESSED*/
D$CNT                BYTE,
DECIMAL              LIT                 '11',
DISPLACEMENT         LIT                 '14',
EL$CNT               LIT                 '6',
HASH$MASK            LIT                 '3FH',
LEVEL                LIT                 '10',
LOCATION             LIT                 '2',
MAX$ID$LEN           LIT                 '15',
NEXT$SYM$ENTRY       BASED NEXT$SYM      ADDRESS,
OCCURS$PTR           ADDRESS             INITIAL(0),
P$LENGTH             LIT                 '3',
REL$ID               LIT                 '5',
SAVE$ADDR            ADDRESS,
S$LENGTH             LIT                 '3',
S$TYPE               LIT                 '2',
START$NAME           LIT                 '13',  /*1 LESS*/
SYMBOL               BASED CUR$SYM(1)    BYTE,
SYMBOL$ADDR          BASED CUR$SYM(1)    ADDRESS,
TEMP$PTR             ADDRESS,
TEMP$ADDR            BASED TEMP$PTR      ADDRESS,
TEMP$BYTE            BASED TEMP$PTR      BYTE;

    /*  *  *  *  *  TYPE LITERALS * * * * * * * * */

DCL
COMP           LIT       '21',
GROUP          LIT        '6',
OCCURS$TYPE    LIT      '128',
RANDOM         LIT        '3',
REL$KEY        LIT       '25',
REL$KEY$UR     LIT       '26',
SEQUENTIAL     LIT        '1',
SEQ$RELATIVE   LIT        '2',
UR$MASK        LIT      '128',
VARIABLE$LENG  LIT        '4';

    /*  *  *  *  SYMBOL TABLE ROUTINES  *  *  *  */

INIT$SYMBOL: PROC;
    /* INITIALIZE HASH TABLE AND FIRST COLLISION FIELD */
    CALL FILL (FREE$STORAGE,0,130);
    NEXT$SYM = FREE$STORAGE + 128;
    NEXT$SYM$ENTRY = 0;
END INIT$SYMBOL;
```

```
GET$P$LENGTH: PROC BYTE;
    RETURN SYMBOL(P$LENGTH);
END GET$P$LENGTH;

SET$ADDRESS: PROC(ADDR);
    DCL ADDR ADDRESS;
    SYMBOL$ADDR(LOCATION) = ADDR;
END SET$ADDRESS;

GET$ADDRESS: PROC ADDRESS;
    RETURN SYMBOL$ADDR(LOCATION);
END GET$ADDRESS;

GET$TYPE: PROC BYTE;
    RETURN SYMBOL(S$TYPE);
END GET$TYPE;

SET$TYPE: PROC(TYPE);
    DCL TYPE BYTE;
    SYMBOL(S$TYPE) = TYPE;
END SET$TYPE;

OR$TYPE: PROC(TYPE);
    DCL TYPE BYTE;
    SYMBOL(S$TYPE) = TYPE OR GET$TYPE;
END OR$TYPE;

GET$LEVEL: PROC BYTE;
    RETURN SYMBOL(LEVEL);
END GET$LEVEL;

SET$LEVEL: PROC (LVL);
    DCL LVL BYTE;
    SYMBOL(LEVEL) = LVL;
END SET$LEVEL;

GET$DECIMAL: PROC BYTE;
    RETURN SYMBOL(DECIMAL);
END GET$DECIMAL;

SET$DECIMAL: PROC (DEC);
    DCL DEC BYTE;
    SYMBOL(DECIMAL) = DEC;
END SET$DECIMAL;

SET$S$LENGTH: PROC(HOW$LONG);
    DCL HOW$LONG ADDRESS;
    SYMBOL$ADDR(S$LENGTH) = HOW$LONG;
END SET$S$LENGTH;
```

```
GET$S$LENGTH: PROC ADDRESS;
    RETURN SYMBOL$ADDR(S$LENGTH);
END GET$S$LENGTH;

SET$ADDR2: PROC (ADDR);
    DCL ADDR ADDRESS;
    SYMBOL$ADDR(ADDR2) = ADDR;
END SET$ADDR2;

SET$TBL$SIZE: PROC(OCCUR);
    DCL OCCUR ADDRESS;
    SYMBOL$ADDR(EL$CNT) = OCCUR;
END SET$TBL$SIZE;

GET$TBL$SIZE: PROC ADDRESS;
    RETURN SYMBOL$ADDR(EL$CNT);
END GET$TBL$SIZE;

SET$IO$ADDRS: PROC;
    SYMBOL$ADDR(LOCATION) = NEXT$SYM;
    SAVE$ADDR = CUR$SYM;
END SET$IO$ADDRS;

GET$PREV$OCCURS:PROC ADDRESS;
    TEMP$PTR = CUR$SYM + STARTNAME + GET$P$LENGTH;
    RETURN TEMP$ADDR;
END GET$PREV$OCCURS;

PROCESS$OCCURS:PROC;
    TEMP$PTR = NEXT$SYM;
    NEXT$SYM = NEXT$SYM + 3;
    TEMP$ADDR = OCCURS$PTR; /*SET PTR TO PREVIOUS OCCURS*/
    CALL OR$TYPE(OCCURS$TYPE);
    TEMP$PTR = TEMP$PTR + 2;
    TEMP$BYTE = D$CNT;
END PROCESS$OCCURS;
    /*  *  *  *  PARSER DECLARATIONS  *  *  *  */
DCL
COMPILING                BYTE       INITIAL(TRUE),
HOLD$LIT(ACCUM$LEN$P$1) BYTE,
HOLD$SYM                 ADDRESS,
ID$STACK(10)             ADDRESS    INITIAL(0),
ID$STACK$PTR             BYTE       INITIAL(0),
INT                      LIT        '67', /* INITIALIZE */
(I,J,K)                  BYTE,
MP                       BYTE,
MPP1                     BYTE,
NOLOOK                   BYTE       INITIAL(TRUE),
REDEF                    BYTE       INITIAL(FALSE),
REDEF$ONE                ADDRESS,
REDEF$TWO                ADDRESS,
```

```
PENDING$LITERAL          BYTE       INITIAL(FALSE),
PENDING$LIT$ID           ADDRESS,
PSTACKSIZE               LIT        '40', /* SIZE OF STACKS */
SCD                      LIT        '70', /* CODE START */
SP                       BYTE       INITIAL(255),
STATE                    BYTE       INITIAL(STARTS),
STATESTACK(PSTACKSIZE)   BYTE,      /* SAVED STATES */
TEMP$HOLD                ADDRESS,
TEMP$TWO                 ADDRESS,
TRUNC$FLAG               BYTE       INITIAL(TRUE),
VALUE(PSTACKSIZE)        ADDRESS,   /* TEMP VALUES */
VALUE$FLAG               BYTE       INITIAL(FALSE),
VALUE$LEVEL              BYTE       INITIAL(0),
VARC(51)                 BYTE;      /*TEMP CHAR STORE*/

    /*  *  *  *  PARSER ROUTINES  *  *  *  *  */

BYTE$OUT: PROC(ONE$BYTE);
    DCL ONE$BYTE BYTE;
    IF NO$CODE THEN RETURN;
    IF (OUTPUT$PTR := OUTPUT$PTR + 1) > OUTPUT$END THEN
        DO
            CALL WRITE$OUTPUT(.OUTPUT$BUFF,.OUTPUT$FCB);
            OUTPUT$PTR=.OUTPUT$BUFF;
        END;
    OUTPUT$CHAR = ONE$BYTE;
END BYTE$OUT;

STRING$OUT: PROC (ADDR,COUNT);
    DCL (ADDR,I,COUNT) ADDRESS, CHAR BASED ADDR BYTE;
    DO I = 1 TO COUNT;
        CALL BYTE$OUT(CHAR);
        ADDR = ADDR+1;
    END;
END STRING$OUT;

ADDR$OUT: PROC(ADDR);
    DCL ADDR ADDRESS;
    CALL BYTE$OUT(LOW(ADDR));
    CALL BYTE$OUT(HIGH(ADDR));
END ADDR$OUT;

FILL$STRING: PROC(COUNT,CHAR);
    DCL (I,COUNT) ADDRESS, CHAR BYTE;
    DO I = 1 TO COUNT;
        CALL BYTE$OUT(CHAR);
    END;
END FILL$STRING;

START$INITIALIZE: PROC(ADDR,CNT);
    DCL (ADDR,CNT) ADDRESS;
```

```
    CALL BYTEOUT(INT);
    CALL ADDR$OUT(ADDR);
    CALL ADDR$OUT(CNT);
END START$INITIALIZE;

BUILD$SYMBOL: PROC(LEN);
    DCL LEN BYTE, TEMP ADDRESS;
    TEMP = NEXT$SYM;
    IF (NEXT$SYM := .SYMBOL(LEN := LEN + DISPLACEMENT))
        > MAX$MEMORY THEN CALL FATAL$ERROR('ST');
    CALL FILL (TEMP,0,LEN);
END BUILD$SYMBOL;
MATCH: PROC ADDRESS;
    /* CHECKS AN IDENTIFIER TO SEE IF IT IS IN THE SYMBOL
    TABLE.  IF IT IS PRESENT, CUR$SYM IS SET FOR ACCESS.
    OTHERWISE A NEW ENTRY IS MADE AND THE PRINT NAME
    IS ENTERED.  ALL NAMES ARE TRUNCATED TO MAX$ID$LEN*/
    DCL POINT ADDRESS,COLLISION BASED POINT ADDRESS,
        (HOLD,I) BYTE;
    IF VARC(0) > MAX$ID$LEN
        THEN VARC(0) = MAX$ID$LEN; /* TRUNCATE IF REQUIRED */
    HOLD = 0;
    DO I = 1 TO VARC(0); /* CALCULATE HASH CODE */
        HOLD = HOLD + VARC(I);
    END;
    POINT = FREE$STORAGE + SHL((HOLD AND HASH$MASK),1);
    UI$FLAG = FALSE;
    DO FOREVER;
        IF COLLISION = 0 THEN
            DO;
                UI$FLAG = TRUE;
                CUR$SYM,COLLISION = NEXT$SYM;
                CALL BUILD$SYMBOL(VARC(0));
                SYMBOL(P$LENGTH) = VARC(0);
                DO I = 1 TO VARC(0);
                    SYMBOL(START$NAME + I) = VARC(I);
                END;
                RETURN CUR$SYM;
            END;
        ELSE
            DO;
                CUR$SYM = COLLISION;
                IF (HOLD := GET$P$LENGTH) = VARC(0) THEN
                    DO;
                        I = 1;
                        DO WHILE
                          SYMBOL(START$NAME + I) =  VARC(I);
                            IF (I := I + 1) > HOLD THEN
                               RETURN (CUR$SYM := COLLISION);
                        END;
                    END;
```

```
            END;
        POINT = COLLISION;
    END;
END MATCH;

ALLOCATE: PROC(BYTES$REQ) ADDRESS;
    DCL (HOLD,BYTES$REQ) ADDRESS;
    HOLD = NEXT$AVAILABLE;
    IF (NEXT$AVAILABLE := NEXT$AVAILABLE + BYTES$REQ)
        > MAX$INT$MEM THEN
        CALL FATAL$ERROR('MO');
    RETURN HOLD;
END ALLOCATE;

DIGIT: PROC(CHAR) BYTE;
    DCL CHAR BYTE;
    RETURN (CHAR <= '9') AND (CHAR >= '0');
END DIGIT;

SET$REDEF: PROC(OLD,NEW);
    DCL (OLD,NEW) ADDRESS;
    REDEF$ONE = OLD;
    REDEF$TWO = NEW;
    REDEF = TRUE;
END SET$REDEF;

SET$CUR$SYM: PROC;
    CUR$SYM = ID$STACK(ID$STACK$PTR);
END SET$CUR$SYM;

STACK$LEVEL: PROC BYTE;
    CALL SET$CUR$SYM;
    RETURN GET$LEVEL;
END STACK$LEVEL;

LOAD$LEVEL: PROC;
    DCL HOLD ADDRESS;

    LOAD$REDEF$ADDR: PROC;
         CUR$SYM = REDEF$ONE;
         HOLD = GET$ADDRESS;
    END LOAD$REDEF$ADDR;

    IF ID$STACK(0) <> 0 THEN
        DO;
            IF VALUE(SP - 2) = 0 THEN
                DO;
                    CALL SET$CUR$SYM;
                    HOLD = GET$S$LENGTH + GET$ADDRESS;
                END;
            ELSE
```

```
                DO;
                  IF FILE$SEC$END THEN
                    DO;
                      IF ID$STACK(ID$STACK$PTR) <> REDEF$ONE
                          THEN
                        DO;
                          CALL PRINT$ERROR('R1');
                          REDEF$ONE = ID$STACK(ID$STACK$PTR);
                        END;
                    END;
                  CALL LOAD$REDEF$ADDR;
                END;
        IF (ID$STACK$PTR := ID$STACK$PTR + 1) > 9 THEN
            DO;
                CALL PRINT$ERROR('EL');
                ID$STACK$PTR = 9;
            END;
    END;
    ELSE HOLD = NEXT$AVAILABLE;
    CUR$SYM,ID$STACK(ID$STACK$PTR) = VALUE(MPP1);
    IF (CUR$SYM<>OCCURS$PTR) AND (D$CNT<>0) THEN
        CALL PROCESS$OCCURS;
    IF (GET$LEVEL = 1) AND (NOT FILE$SEC$END) THEN
        CALL SET$ADDR2(SAVE$ADDR);
    CALL SET$ADDRESS(HOLD);
END LOAD$LEVEL;

REDEF$OR$VALUE: PROC;
    DCL (HOLD,HOLD1,TEMP) ADDRESS,
        (CHAR,LVL$NBR) BYTE;
    IF REDEF THEN
        DO;
            IF REDEF$TWO = CUR$SYM THEN
                DO;
                    HOLD = GET$S$LENGTH;
                    LVL$NBR = GET$LEVEL;
                    CUR$SYM = REDEF$ONE;
                    IF HOLD <> (HOLD1 := GET$S$LENGTH) THEN
                      DO;
                        IF (LVL$NBR = 1)
                        AND (NOT FILE$SEC$END) THEN
                            DO;
                                CUR$SYM = SAVE$ADDR;
                                CALL SET$TYPE(VARIABLE$LENG);
                                IF HOLD>HOLD1 THEN
                                    CALL SET$S$LENGTH(HOLD);
                                ELSE
                                    CALL SET$S$LENGTH(HOLD1);
                            END;
                        IF HOLD > HOLD1 THEN
                          DO;
```

```
                        IF LVL$NBR = 1 THEN
                          TEMP = ALLOCATE(HOLD - HOLD1);
                        ELSE
                          DO;
                            CALL PRINT$ERROR('R2');
                            CUR$SYM = REDEF$TWO;
                            CALL SET$S$LENGTH(HOLD);
                          END;
                      END;
                  END; /* END IF HOLD <> */
            END; /* END IF REDEF$TWO = CUR$SYM */
    END; /* END IF REFEF */
ELSE IF PENDING$LITERAL = 0 THEN RETURN;
IF (PENDING$LIT$ID<>ID$STACK$PTR) OR VALUE$FLAG THEN
    RETURN;
IF PENDING$LITERAL <> 0 THEN
    CALL START$INITIALIZE(GET$ADDRESS,HOLD :=
        GET$S$LENGTH);
IF PENDING$LITERAL > 2 THEN
    DO;
        IF PENDING$LITERAL = 3 THEN CHAR = '0';
        ELSE IF PENDING$LITERAL = 4 THEN CHAR = ' ';
        ELSE IF PENDING$LITERAL = 5 THEN CHAR = QUOTE;
        CALL FILL$STRING(HOLD,CHAR);
    END;
ELSE IF PENDING$LITERAL = 2 THEN
    DO;
        IF HOLD <= HOLD$LIT(0) THEN
            CALL STRING$OUT(.HOLD$LIT(1),HOLD);
        ELSE
          DO;
             CALL STRING$OUT(.HOLD$LIT(1),HOLD$LIT(0));
             CALL FILL$STRING(HOLD - HOLD$LIT(0),' ');
           END;
    END;
ELSE IF PENDING$LITERAL = 1 THEN
    DO;
        DCL (H$DEC,H$LENGTH,H,L,L$DEC,L$LENGTH,SIGN,TYPE)
            BYTE, TEMP(20) BYTE, ZONE LIT '80H';
        IF ((TYPE := GET$TYPE) < 16 OR (TYPE > 21 THEN
            CALL PRINT$ERROR('NV');
        L$LENGTH = GET$LENGTH;
        L$DEC = L$LENGTH - GET$DECIMAL;
        IF TYPE = 20 THEN L$DEC = L$DEC 0 1;
        H$LENGTH = HOLD$LIT(0);
        H$DEC = H$LENGTH + 1;
        SIGN = '+';
        IF HOLD$LIT(1) = '-' THEN
            SIGN = '-';
        DO H = 1 TO H$LENGTH;
            IF HOLD$LIT(H) = '.' THEN H$DEC = H;
```

```
END;
DO L = 0 TO 19;
    TEMP(L) = '0';
END;
L = L$DEC - 1;
H = H$DEC;
DO WHILE (((L := L + 1) < L$LENGTH) AND
          ((H := H + 1) <= H$LENGTH));
    TEMP(L) = HOLD$LIT(H);
END;
L = L$DEC;
H = H$DEC;
DO WHILE (((L := L - 1) < 255) AND
          ((H := H - 1) > 0) AND
           (HOLD$LIT(H) <> SIGN));
    TEMP(L) = HOLD$LIT(H);
END;
IF ((H > 1) OR
   ((H = 1) AND (HOLD$LIT(1) <> SIGN))) THEN
    CALL PRINT$ERROR('SL');
IF SIGN = '-' THEN
    IF TYPE = 17 THEN
        TEMP(0) = TEMP(0) OR ZONE;
    ELSE IF TYPE = 18 THEN
        TEMP(L$LENGTH) = TEMP(L$LENGTH) OR ZONE;
IF TYPE = 19 THEN
    DO;
        IF TEMP(0) <> '0' THEN
            CALL PRINT$ERROR('SL');
        TEMP(0) = SIGN;
    END;
ELSE IF TYPE = 20 THEN
    TEMP(L$LENGTH - 1) = SIGN;
IF TYPE = 21 THEN
    DO;
        IF SIGN = '+' THEN
            TEMP(L$LENGTH) = '0';
        ELSE TEMP(L$LENGTH) = '1';
        IF (L$LENGTH MOD 2) THEN L = 0;
        ELSE
            DO;
                CALL BYTE$OUT(TEMP(0) - 30H);
                L = 1;
            END;
        DO WHILE L < L$LENGTH;
            CALL BYTE$OUT(SHL((TEMP(L) - 30H),4)
                OR (TEMP(L + 1) - 30H));
            L = L + 2;
        END;
        DO I = L$LENGTH / 2 + 2 TO L$LENGTH;
            CALL BYTE$OUT(00H):
```

```
                    END;
                END;
            ELSE CALL STRING$OUT(.TEMP,L$LENGTH);
        END;
    IF NOT VALUE$FLAG THEN PENDING$LITERAL = 0;
END REDEF$OR$VALUE;

REDUCE$STACK: PROC;
    DCL HOLD$LENGTH ADDRESS;
    CALL SET$CUR$SYM;
    CALL REDEF$OR$VALUE;
    HOLD$LENGTH = GET$S$LENGTH;
    IF GET$TYPE > OCCURS$TYPE AND GET$TPL$SIZE <> 0 THEN
        DO;
            HOLD$LENGTH=HOLD$LENGTH * GET$TBL$SIZE;
            IF (D$CNT := D$CNT - 1) <> 0 THEN
                OCCURS$PTR = GET$PREV$OCCURS;
            ELSE OCCURS$PTR = 0;
        END;
    ID$STACK$PTR=ID$STACK$PTR - 1;
    CALL SET$CUR$SYM;
    CALL SET$S$LENGTH(GET$S$LENGTH + HOLD$LENGTH);
    CALL OR$TYPE(GROUP);
END REDUCE$STACK;

END$OF$RECORD: PROC;
    DO WHILE ID$STACK$PTR <> 0;
        CALL SET$CUR$SYM;
        CALL REDEF$OR$VALUE;
        ID$STACK(ID$STACK$PTR) = 0;
        ID$STACK$PTR = ID$STACK$PTR - 1;
    END;
    CALL SET$CUR$SYM;
    CALL REDEF$OR$VALUE;
    ID$STACK(0) = 0;
    TEMP$HOLD = ALLOCATE(GET$S$LENGTH);
END  END$OF$RECORD;

CONVERT$INTEGER: PROC;
    DCL INTEGER ADDRESS;
    INTEGER = 0;
    DO I = 1 TO VARC(0);
        IF NOT DIGIT(VARC(I)) THEN CALL PRINT$ERROR('NN');
        INTEGER = SHL(INTEGER,3) + SHL(INTEGER,1) +
            (VARC(I) - '0');
    END;
    VALUE(SP) = INTEGER;
END CONVERT$INTEGER;

OR$VALUE: PROC(PTR,ATTRIB);
    DCL PTR BYTE, ATTRIB ADDRESS;
```

```
    VALUE(PTR) = VALUE(PTR) OR ATTRIB;
END OR$VALUE;

BUILD$FCB: PROC;
    DCL TEMP ADDRESS;
    DCL BUFFER(12) BYTE,(CHAR,I,J) BYTE;
    CALL FILL(.BUFFER,' ',12);
    IF VARC(2) = ':' THEN
        DO;
            BUFFER(0) = VARC(1) AND 0FH;
            I = 2;
        END;
    ELSE
        DO;
            BUFFER(0) = 0;
            I = 0;
        END;
    J = 1;
    DO WHILE (J < 12) AND (I< VARC(0));
        IF (CHAR := VARC(I := I + 1)) = '.' THEN J = 9;
        ELSE DO;
            BUFFER(J) = CHAR;
            J = J + 1;
        END;
    END;
    CALL SET$ADDR2(TEMP := ALLOCATE(165));
    CALL START$INITIALIZE(TEMP,37);
    CALL STRING$CUT(.BUFFER,12);
    CALL FILL$STRING(25,0);
    CALL OR$VALUE(SP - 1,1);
END BUILD$FCB;

SET$SIGN: PROC(NUMB);
    DCL NUMB BYTE;
    IF GET$TYPE = 17 THEN CALL SET$TYPE(VALUE(SP) + NUMB);
    ELSE CALL PRINT$ERROR('SG');
    IF VALUE(SP) <> 0 THEN
        CALL SET$S$LENGTH(GET$S$LENGTH + 1);
END SET$SIGN;

NUM$TRUNC: PROC;
    DCL (I,J,TRUNC$TYPE,TRUNC$ZERO,SIGN$FLAG,DEC$FLAG) BYTE;
    TRUNC$ZERO = TRUE;
    SIGN$FLAG,DEC$FLAG = FALSE;
    HOLD$LIT(0),I = 0;
    J = 1;
    IF ((TRUNC$TYPE := GET$TYPE) >= 16)
                AND (TRUNC$TYPE <= 21) THEN
      DO WHILE J <= VARC(0);
        IF (VARC(J) <> '+') AND (VARC(J) <> '-') THEN
          DO;
```

```
            IF (VARC(J) = '0') AND TRUNC$ZERO THEN J = J;
            ELSE IF ((VARC(J) >= '0') AND (VARC(J) <= '9'))
                   OR (VARC(J) = '.') THEN
              DO;
                IF DEC$FLAG AND (VARC(J) = '.') THEN
                  CALL PRINT$ERROR('MD');
                ELSE DO;
                  HOLD$LIT(HOLD$LIT(0) := HOLD$LIT(0) + 1) =
                      VARC(J);
                  IF VARC(J) <> '0' THEN TRUNC$ZERO = FALSE;
                  IF VARC(J) = '.' THEN DEC$FLAG = TRUE;
                  I = I + 1;
                END;
              END;
            ELSE IF ((VARC(J) < '0') OR (VARC(J) > '9')) AND
              (VARC(J) <> '.') THEN CALL PRINT$ERROR('NN');
          END;
        ELSE IF SIGN$FLAG THEN CALL PRINT$ERROR('MS');
        ELSE IF (VARC(J) = '+') OR (VARC(J) = '-') THEN
            DO;
                IF TRUNC$TYPE = 16 THEN
                    CALL PRINT$ERROR('SG');
                ELSE
                    DO;
                        HOLD$LIT(HOLD$LIT(0) :=
                            HOLD$LIT(0) + 1)=VARC(J);
                        SIGN$FLAG = TRUE;
                        I = I + 1;
                    END;
            END;
        J = J + 1;
      END;/* DO WHILE LOOP */
    HOLD$LIT(0) = I;
    IF ((HOLD$LIT(0) = 1) AND ((HOLD$LIT(1) = '+') OR
       (HOLD$LIT(1) = '-'))) OR (HOLD$LIT(0) = '0') THEN
      HOLD$LIT(0),HOLD$LIT(1) = 0;
END NUM$TRUNC;

PIC$ANALIZER: PROC;
    DCL   /* WORK AREAS AND VARIABLES */
    BUFFER(133) BYTE,
    CHAR        BYTE,
    COUNT       ADDRESS,
    DEC$COUNT   BYTE,
    DEC$FLAG    BYTE,
    DIGITS      BYTE,
    FLAG        BYTE,
    FLAGS(3)    BYTE,
    FLOAT$PSIT  BYTE,
    FLOAT$VALUE BYTE,
    I           BYTE,
```

```
J             ADDRESS,
K             BYTE,
REPITITIONS ADDRESS,
SAVE          BYTE,
TEMP          ADDRESS,
TYPE          BYTE,

/* * * MASKS * * */
ALPHA     LIT    '1',
A$EDIT    LIT    '2',
A$N       LIT    '4',
EDIT      LIT    '8',
NUM       LIT   '16',
NUM$EDIT  LIT   '32',
DEC       LIT   '64',
SIGNED    LIT  '128',

A$E$MASK        LIT        '11111100B',
A$N$MASK        LIT        '11101010B',
A$N$E$MASK      LIT        '11100000B',
ALPHA$MASK      LIT        '11111110B',
NUM$MASK        LIT        '10101111B',
NUM$ED$MASK     LIT        '10000101B',
S$NUM$MASK      LIT        '00101111B',

/* TYPES */
ATYPE    LIT    '8',
AETYPE   LIT   '72',
ANTYPE   LIT    '9',
ANETYPE  LIT   '73',
NTYPE    LIT   '16',
NETYPE   LIT   '80',
SNTYPE   LIT   '17';

INC$COUNT: PROC(SWITCH);
     DCL SWITCH BYTE;
     FLAG = FLAG OR SWITCH;
     IF (COUNT := COUNT + 1) < 133 THEN
         BUFFER(COUNT) = CHAR;
END INC$COUNT;

CHECK: PROC (MASK) BYTE;
     DCL MASK BYTE;
     RETURN NOT ((FLAG AND MASK) <> 0);
END CHECK;

PIC$ALLOCATE: PROC(AMT) ADDRESS;
     DCL AMT ADDRESS;
     IF (MAX$INT$MEM := MAX$INT$MEM - AMT)
          < NEXT$AVAILABLE THEN CALL FATAL$ERROR ('MO');
     RETURN MAX$INT$MEM;
```

```
END PIC$ALLOCATE;

SIGN: PROC(CHAR) BYTE;
    DCL CHAR BYTE;
    RETURN (CHAR = '+') OR (CHAR = '-');
END SIGN;

FLOAT$CHECK: PROC(I);
    DCL I BYTE;
    IF FLOAT$VALUE = Ø AND FLAGS(I) THEN
        FLOAT$VALUE = CHAR;
    IF CHAR <> FLOAT$VALUE AND FLAGS(I) THEN
        CALL PRINT$ERROR('P1');
    IF FLAGS(I) THEN
        DO;
            FLOAT$PSIT = COUNT + 1;
            DIGITS = DIGITS + 1;
        END;
    ELSE
        FLAGS(I) = TRUE;
    CALL INC$COUNT(NUM$EDIT);
END FLOAT$CHECK;

/* PROCEDURE EXECUTION STARTS HERE */

CUR$SYM = HOLD$SYM;
IF (GET$LEVEL = VALUE$LEVEL) THEN VALUE$FLAG = FALSE;
DEC$FLAG,FLAGS(Ø),FLAGS(1) = FALSE;
FLAGS(2) = TRUE;
COUNT,DEC$COUNT,DIGITS,FLAG,FLOAT$VALUE,TYPE = Ø;
/* CHECK FOR EXCESSIVE LENGTH */
IF VARC(Ø) > 3Ø THEN
    DO;
        CALL PRINT$ERROR('PC');
        RETURN;
    END;
/* SET FLAG BITS AND COUNT LENGTH */
I  = 1;
DO WHILE I <= VARC(Ø);
     IF (CHAR := VARC(I)) = 'A' THEN
         CALL INC$COUNT(ALPHA);
     ELSE IF CHAR  = 'B' THEN CALL INC$COUNT(A$EDIT);
     ELSE IF CHAR  = '9' THEN
         DO;
             DIGITS = DIGITS + 1;
             CALL INC$COUNT(NUM);
         END;
     ELSE IF CHAR  = 'X' THEN CALL INC$COUNT(A$N);
     ELSE IF (CHAR = 'S') AND (COUNT=Ø) THEN
            FLAG = FLAG OR SIGNED;
     ELSE IF (CHAR = 'V') AND (DEC$COUNT=Ø) THEN
```

```
        DO;
            FLAG = FLAG OR DEC;
            DEC$COUNT = COUNT;
            DEC$FLAG = TRUE;
        END;
ELSE IF(CHAR = '/') OR (CHAR='0') THEN
         CALL INC$COUNT(EDIT);
ELSE IF CHAR = '$' THEN CALL FLOAT$CHECK(0);
ELSE IF SIGN(CHAR) THEN CALL FLOAT$CHECK(1);
ELSE IF (CHAR = '*') OR (CHAR = 'Z') THEN
    CALL FLOAT$CHECK(2);
ELSE IF CHAR = ',' THEN CALL INC$COUNT(NUM$EDIT);
ELSE IF (CHAR = '.') AND (DEC$COUNT=0) THEN
        DO;
            CALL INC$COUNT(NUM$EDIT);
            DEC$COUNT = COUNT;
            DEC$FLAG = TRUE;
        END;
ELSE IF ((CHAR = 'C' AND VARC(I + 1)='R') OR
     (CHAR = 'D' AND VARC(I + 1)='B')) AND
     I = VARC(0) - 1 AND NOT FLAGS(1) THEN
         DO;
             CALL INC$COUNT(NUM$EDIT);
             CHAR = VARC(I:=I + 1);
             CALL INC$COUNT(NUM$EDIT);
             IF NOT DEC$FLAG THEN
                 DO;
                     DEC$COUNT = VARC(0) - 1;
                     DEC$FLAG = TRUE;
                 END;
         END;
ELSE IF (CHAR = '(') AND (COUNT<>0) THEN
    DO;
        SAVE = VARC(I - 1);
        REPITITIONS = 0;
        DO WHILE (CHAR := VARC(I := I + 1)) <> ')';
            IF CHAR < '0' OR CHAR > '9' THEN
                CALL PRINT$ERROR('P2');
            REPITITIONS = SHL(REPITITIONS,3)  +
            SHL(REPITITIONS,1)  + (CHAR  - '0');
        END;
        CHAR = SAVE;
        IF REPITITIONS <> 0 THEN
            DO;
                DO J = 1 TO REPITITIONS - 1;
                   CALL INC$COUNT(0);
                END;
                IF SIGN(SAVE) OR SAVE = '$'
                 OR SAVE = 'Z' OR SAVE = '9'
                 OR SAVE = '*' THEN
                  DIGITS = DIGITS + REPITITIONS - 1;
```

```
                END;
            ELSE
                COUNT = COUNT - 1;
        END;
    ELSE DO;
            CALL PRINT$ERROR('P3');
            RETURN;
        END;
    I = I + 1;
END; /* END OF DO WHILE I <=  VARC */
IF NOT DEC$FLAG AND SIGN(VARC(I - 1)) THEN
    DO;
        DEC$COUNT = VARC(0);
        DEC$FLAG = TRUE;
    END;
/* AT THIS POINT THE TYPE CAN BE DETERMINED */
IF      CHECK(NUM$MASK)   THEN TYPE = NTYPE;
ELSE IF CHECK(SNUM$MASK)  THEN TYPE = SNTYPE;
ELSE IF CHECK(ALPHA$MASK) THEN TYPE = ATYPE;
ELSE IF CHECK(A$E$MASK)   THEN TYPE = AETYPE;
ELSE IF CHECK(A$N$MASK)   THEN TYPE = ANTYPE;
ELSE IF CHECK(A$N$E$MASK)     AND (((FLAG AND 06H) <> 0)
    OR ((FLAG AND 09H) <> 0) OR   ((FLAG AND 12H) <> 0))
                          THEN TYPE = ANETYPE;
ELSE IF CHECK(NUM$ED$MASK) THEN
    DO;
        TYPE = NETYPE;
        IF FLOAT$VALUE <> 0 THEN
            DO;
                I = 1;
                DO WHILE VARC(I) <> FLOAT$VALUE;
                    I = I + 1;
                END;
                DO I = I + 1 TO FLOAT$PSIT;
                    IF VARC(I) <> FLOAT$VALUE AND
                       VARC(I) <> 'B' AND
                       VARC(I) <> '/' AND
                       VARC(I) <> '0' AND
                       VARC(I) <> ',' THEN
                            DO;
                                CALL PRINT$ERROR('P4');
                                I = FLOAT$PSIT;
                            END;
                END;
            END;
    END;
IF TYPE = 0 THEN CALL PRINT$ERROR('P5');
ELSE DO;
        IF (GET$TYPE = 128) THEN
            CALL SET$TYPE(128 + TYPE);
        ELSE CALL SET$TYPE(TYPE);
```

```
            CALL SET$SLENGTH(COUNT + GET$S$LENGTH);
            IF (TYPE AND 64) <> 0 THEN
                DO;
                    CALL SET$ADDR2(TEMP :=
                        PIC$ALLOCATE(COUNT));
                    CALL START$INITIALIZE(TEMP,COUNT);
                    CALL STRING$OUT(.BUFFER + 1,COUNT);
                END;
            IF DIGITS > 18 THEN
                CALL PRINT$ERROR('P6');
            IF DEC$FLAG THEN
                CALL SET$DECIMAL(COUNT - DEC$COUNT);
        END;
   IF (NOT TRUNC$FLAG) AND ((TYPE = 16) OR (TYPE = 17)) THEN
       DO;
           DO K = 0 TO HOLD$LIT(0);
             VARC(K) = HOLD$LIT(K);
           END;
           CALL NUM$TRUNC;
           TRUNC$FLAG = TRUE;
       END;
END PIC$ANALIZER;

SET$FILE$ATTRIB: PROC;
    DCL TEMP ADDRESS, TYPE BYTE;
    IF CUR$SYM <> VALUE(MPP1) THEN
        DO;
            TEMP = CUR$SYM;
            CUR$SYM = VALUE(MPP1);
            SYMBOL$ADDR(REL$ID) = TEMP;
        END;
    IF NOT (TEMP := VALUE(SP - 1)) THEN
        CALL PRINT$ERROR('NF');
    ELSE  DO;
             IF (TEMP = 1) OR (TEMP=5) THEN TYPE=SEQUENTIAL;
             ELSE IF TEMP = 15 THEN TYPE=RANDOM;
             ELSE IF TEMP = 13 THEN TYPE=SEQ$RELATIVE;
             ELSE DO;
                      CALL PRINT$ERROR('IA');
                      TYPE = 1;
                  END;
          END;
    CALL SET$TYPE(TYPE + UR$MASK);
END SET$FILE$ATTRIB;

LOAD$LITERAL: PROC(LIT$ONE);
    DCL (I,LIT$ONE,LIT$TYPE) BYTE;
    LIT$TYPE = GET$TYPE;
    IF LIT$TYPE <> 0 THEN VALUE$FLAG = FALSE;
    ELSE DO;
             VALUE$FLAG = TRUE;
```

```
            VALUE$LEVEL = GET$LEVEL;
        END;
    IF PENDING$LITERAL <> Ø THEN CALL PRINT$ERROR ('LE');
    ELSE IF (LIT$ONE = Ø) OR (LIT$TYPE = Ø) THEN
        DO;
            DO I = Ø TO VARC(Ø);
                HOLD$LIT(I) = VARC(I);
            END;
            IF (LIT$ONE = 1) AND (LIT$TYPE = Ø) THEN
                TRUNC$FLAG = FALSE;
        END;
    ELSE IF (LIT$ONE = 1) AND ((LIT$TYPE >= 16) AND
        (LIT$TYPE <= 21)) THEN
            CALL NUM$TRUNC;
    ELSE IF (LIT$ONE = 1) AND (LIT$TYPE <> Ø) THEN
                DO;
                    CALL PRINT$ERROR('LV');
                    DO I = Ø TO VARC(Ø);
                        HOLD$LIT(I) = VARC(I);
                    END;
                    PENDING$LITERAL = 2;
                END;
END LOAD$LITERAL;

REDEF$TEST:  PROC;
     DCL  SAVE$REDEF  BYTE,
          (SAVE$REDEF$ONE,SAVE$RFDEF$TWO)  ADDRESS;
     SAVE$REDEF$ONE = REDEF$ONE;
     SAVE$REDEF$TWO = REDEF$TWO;
     REDEF$ONE = CUR$SYM;
     CALL SET$CUR$SYM;
     IF (GET$TYPE > OCCURS$TYPE) AND (GET$TBL$SIZE <> Ø) THEN
         IF (D$CNT := D$CNT - 1) <> Ø THEN
             OCCURS$PTR = GET$PREV$OCCURS;
         ELSE OCCURS$PTR = Ø;
     REDEF$TWO = CUR$SYM;
     SAVE$REDEF = REDEF;
     REDEF = TRUE;
     CALL REDEF$OR$VALUE;
     ID$STACK(ID$STACK$PTR) = Ø;
     ID$STACK$PTR = ID$STACK$PTR - 1;
     REDEF$ONE = SAVE$REDEF$ONE;
     REDEF$TWO = SAVE$REDEF$TWO;
     REDEF = SAVE$REDEF;
END REDEF$TEST;

CHECK$LVL$FILES:  PROC;
     DCL NEW$LEVEL  BYTE;
     HOLD$SYM,CUR$SYM = VALUE(MP - 1);
     CALL SET$LEVEL(NEW$LEVEL := VALUE(MP - 2));
     IF NEW$LEVEL = 1 THEN
```

```
        DO;
        IF ID$STACK(0) <> 0 THEN
          DO;
            DO WHILE STACK$LEVEL > 1;
              CALL REDUCE$STACK;
            END;
            DO WHILE ID$STACK$PTR <> 0;
              CALL SET$CUR$SYM;
              CALL REDEF$OR$VALUE;
              ID$STACK(ID$STACK$PTR) = 0;
              ID$STACK$PTR = ID$STACK$PTR - 1;
            END;
            CUR$SYM = HOLD$SYM;
            CALL SET$REDEF(ID$STACK(0),VALUE(MP - 1));
            VALUE(MP) = 1; /* SET REDEFINE FLAG */
          END;
        END;
      ELSE DO WHILE STACK$LEVEL >= NEW$LEVEL;
             CALL REDUCE$STACK;
           END;
END CHECK$LVL$FILES;

CHECK$LVL$WORK: PROC;
    DCL NEW$LEVEL       BYTE,
        SAVE$SYM$LVL    BYTE,
        STACK$REDUCED   BYTE,
        SAVE$REDEF      BYTE,
        REDEF$FLAG      BYTE, /*NXT LVL IS A REDEFINES*/
        SAVE$SYM        ADDRESS;

SET$VALUE$CLAUSE: PROC;
    SAVE$REDEF = REDEF;
    REDEF = FALSE;
    CALL SET$CUR$SYM;
    CALL REDEF$OR$VALUE;
    REDEF = SAVE$REDEF;
    CUR$SYM = HOLD$SYM;
END SET$VALUE$CLAUSE;

    TRUNC$FLAG = TRUE;
    STACK$REDUCED = FALSE;
    HOLD$SYM,CUR$SYM = VALUE(MP - 1);
    CALL SET$LEVEL(NEW$LEVEL := VALUE(MP - 2));
    REDEF$FLAG = VALUE(MP); /*SET IN PROD #64*/
    IF NEW$LEVEL = 1 OR NEW$LEVEL = 77 THEN
      DO;
        IF STACK$LEVEL = 77 THEN
          CALL END$OF$RECORD;
        ELSE
          DO;
            DO WHILE STACK$LEVEL > 1
```

```
            AND ID$STACK(ID$STACK$PTR) <> 0;
          SAVE$SYM,CUR$SYM = ID$STACK(ID$STACK$PTR - 1);
          SAVE$SYM$LVL = GET$LEVEL;
          IF SAVE$SYM$LVL = STACK$LEVEL THEN
            DO;
            CUR$SYM = SAVE$SYM;
            CALL REDEF$TEST;
            END;
          ELSE IF STACK$LEVEL > 1 THEN
            DO;
              CALL REDUCE$STACK;
              IF VALUE$FLAG
                AND (VALUE$LEVEL = STACK$LEVEL) THEN
                DO;
                  VALUE$FLAG = FALSE;
                  CALL SET$VALUE$CLAUSE;
                END;
            END;
        END;/* DO WHILE LOOP */
        IF STACK$LEVEL = 1 AND ID$STACK$PTR <> 0 THEN
          DO;
            CUR$SYM = ID$STACK(ID$STACK$PTR - 1);
            CALL REDEF$TEST;
          END;
        IF REDEF$FLAG = 0
          AND ID$STACK(ID$STACK$PTR) <> 0 THEN
          DO;
            CALL END$OF$RECORD;
            REDEF = FALSE;
          END;
        IF (REDEF$FLAG = 1)
          AND (ID$STACK(ID$STACK$PTR) = REDEF$ONE)
            THEN CALL SET$VALUE$CLAUSE;
      END;
  END;
ELSE IF STACKLEVEL = 77 THEN CALL PRINT$ERROR('L7');
ELSE IF STACK$LEVEL >= NEW$LEVEL THEN
  DO;
    IF (STACK$LEVEL = NEW$LEVEL) AND (REDEF$FLAG = 1) AND
       (ID$STACK(ID$STACK$PTR) = REDEF$ONE) THEN
       CALL SET$VALUE$CLAUSE;
    DO WHILE NOT STACK$REDUCED;
      SAVE$SYM,CUR$SYM = ID$STACK(ID$STACK$PTR - 1);
      SAVE$SYM$LVL = GET$LEVEL;
      IF SAVE$SYM$LVL = STACK$LEVEL THEN
        DO;
          CUR$SYM = SAVE$SYM;
          CALL REDEF$TEST;
        END;
      ELSE IF (STACK$LEVEL >= NEW$LEVEL)
        AND (REDEF$FLAG = 0) THEN
```

```
            DO;
              CALL REDUCE$STACK;
              IF VALUE$FLAG AND (VALUE$LEVEL = STACK$LEVEL)
                AND (VALUE$LEVEL = NEW$LEVEL) THEN
                DO;
                  VALUE$FLAG = FALSE;
                  CALL SET$VALUE$CLAUSE;
                END;
              IF STACK$LEVEL<NEW$LEVEL THEN
                  STACK$REDUCED = TRUE;
            END;
          ELSE IF (STACK$LEVEL >= NEW$LEVEL)
            AND (REDEF$FLAG = 1) THEN
            DO;
              IF STACK$LEVEL>NEW$LEVEL THEN
                  CALL REDUCE$STACK;
              IF VALUE$FLAG
                AND (VALUE$LEVEL = STACK$LEVEL) THEN
                DO;
                  VALUE$FLAG = FALSE;
                  CALL SET$VALUE$CLAUSE;
                END;
                IF STACK$LEVEL <= NEW$LEVEL THEN
                  STACK$REDUCED = TRUE;
              END;
        END; /* DO WHILE LOOP */
      END;
    CUR$SYM = HOLD$SYM;
END CHECK$LVL$WORK;

CODE$GEN: PROC(PRODUCTION);
    DCL PRODUCTION BYTE,
        LIT$TYPE   BYTE;
    IF PRINT$PROD THEN
        DO;
            CALL CRLF;
            CALL PRINTCHAR(POUND);
            CALL PRINT$NUMBER(PRODUCTION);
        END;

    DO CASE PRODUCTION;

  /*  P R O D U C T I O N S */

 /* CASE Ø NOT USED                                                  */
            ;
 /*       1    <PROGRAM> ::= <ID - DIV> <E - DIV> <D - DIV>        */
 /*       1                  PROCEDURE                              */
    DO;
      COMPILING = FALSE;
      CALL DISPLAY$LINE;
```

```
   END;
/*      2   <ID - DIV> ::= IDENTIFICATION DIVISION .          */
/*                         PROGRAM-ID .                       */
/*      2                  <COMMENT> . <ID-LIST>              */
   ;    /* NO ACTION REQUIRED */
/*       3   <ID-LIST> ::= <AUTH> <INS> <DATE> <SEC>          */
  ;     /* NO ACTION REQUIRED */
/*      4   <AUTH> ::= AUTHOR . <COMMENT> .                   */
   ;    /* NO ACTION REQUIRED */
/*      5              \! <EMPTY>                             */
   ;    /* NO ACTION REQUIRED */
/*      6   <INS> ::= INSTALLATION . <COMMENT> .              */
   ;    /* NO ACTION REQUIRED */
/*      7            \! <EMPTY>                               */
   ;    /* NO ACTION REQUIRED */
/*      8   <DATE> ::= DATE - WRITTEN . <COMMENT> .           */
   ;    /* NO ACTION REQUIRED */
/*      9            \! <EMPTY>                               */
   ;    /* NO ACTION REQUIRED */
/*     10   <SEC> ::= SECURITY . <COMMENT> .                  */
   ;    /* NO ACTION REQUIRED */
/*     11            \! <EMPTY>                               */
   ;    /* NO ACTION REQUIRED */
/*     12   <COMMENT> ::= <INPUT>                             */
   ;    /* NO ACTION REQUIRED */
/*     13            \! <COMMENT> <INPUT>                     */
   ;    /* NO ACTION REQUIRED */
/*     14   <E - DIV> ::= ENVIRONMENT DIVISION .              */
/*                        CONFIGURATION SECTION .             */
/*     14                 <SRC - OBJ> <I - O>                 */
   ;    /* NO ACTION REQUIRED  */
/*     15              \! <EMPTY>                             */
  ;     /* NO ACTION REQUIRED */
/*     16   <SRC - OBJ> ::= SOURCE - COMPUTER . <COMMENT>     */
/*                          <DEBUG> .                         */
/*     16                 OBJECT - COMPUTER . <COMMENT> .     */
   ;    /* NO ACTION REQUIRED */
/*     17   <DEBUG> ::= DEBUGGING MODE                        */
   DEBUGGING = TRUE;    /* SETS A SCANNER TOGGLE */
/*     18            \! <EMPTY>                               */
   ;    /* NO ACTION REQUIRED */
/*     19   <I-O> ::= INPUT-OUTPUT SECTION . FILE-CONTROL     */
/*     19              . <FILE - CONTROL - LIST> <IC>         */
   ;    /* NO ACTION REQUIRED */
/*     20            \! <EMPTY>                               */
   ;    /* NO ACTION REQUIRED */
/*     21   <FILE-CONTROL-LIST> ::= <FILE-CONTROL-ENTRY>      */
   ;    /* NO ACTION REQUIRED */
/*     22                          \! <FILE-CONTROL-LIST>     */
/*     22                            <FILE-CONTROL-ENTRY>     */
   ;    /* NO ACTION REQUIRED */
```

```
/*     23 <FILE-CONTROL-ENTRY> ::= SELECT <ID>                    */
/*                                   <ATTRIBUTE-LIST> .          */
   CALL SET$FILE$ATTRIB;
/*     24   <ATTRIBUTE-LIST> ::= <ONE-ATTRIB>                    */
   ;    /* NO ACTION REQUIRED */
/*     25                      \! <ATTRIBUTE-LIST>               */
/*                                <ONE-ATTRIB>                   */
   VALUE(MP) = VALUE(SP) OR VALUE(MP);
/*     26   <ONE-ATTRIB> ::= ORGANIZATION <ORG-TYPE>             */
   VALUE(MP) = VALUE(SP);
/*     27                   \! ACCESS <ACC-TYPE> <RELATIVE>      */
   VALUE(MP) = VALUE(MPP1) OR VALUE(SP);
/*     28                   \! ASSIGN <INPUT>                    */
   CALL BUILD$FCB;
/*     29   <ORG-TYPE> ::= SEQUENTIAL                            */
   ;    /* NO ACTION REQUIRED - DEFAULT */
/*     30                  \! RELATIVE                           */
   CALL OR$VALUE(SP,4);
/*     31                  \! INDEXED                            */
   CALL PRINT$ERROR('NI');
/*     32   <ACC-TYPE> ::= SEQUENTIAL                            */
   ;    /* NO ACTION REQUIRED - DEFAULT */
/*     33                  \! RANDOM                             */
   CALL OR$VALUE(SP,2);
/*     34   <RELATIVE> ::= RELATIVE <ID>                         */
   DO;
       CALL OR$VALUE(MP,8);
       CURSYM = VALUE(SP);
       CALL SET$TYPE(REL$KEY$UR);
   END;
/*     35                  \! <EMPTY>                            */
   ;    /* NO ACTION REQUIRED - DEFAULT */
/*     36   <IC> ::= I-O-CONTROL . <SAME-LIST>                   */
   ;    /* NO ACTION REQUIRED */
/*     37          \! <EMPTY>                                    */
   ;    /* NO ACTION REQUIRED */
/*     38   <SAME - LIST> ::= <SAME - ELEMENT>                   */
   ;    /* NO ACTION REQUIRED */
/*     39                  \! <SAME - LIST> <SAME - ELEMENT>*/
   ;    /* NO ACTION REQUIRED */
/*     40   <SAME-ELEMENT> ::= SAME <ID-STRING> .                */
   ;    /* NO ACTION REQUIRED */
/*     41   <ID-STRING> ::= <ID>                                 */
   ;    /* NO ACTION REQUIRED */
/*     42                  \! <ID-STRING> <ID>                   */
   ;    /* NO ACTION REQUIRED */
/*     43   <D-DIV> ::= DATA DIVISION . <FILE-SECTION>           */
/*                      <WORK>                                   */
/*     43               <LINK>                                   */
   ;    /* NO ACTION REQUIRED  */
/*     44   <FILE-SECTION> ::= FILE SECTION . <FILE-LIST> */
```

```
   FILE$SEC$END = TRUE;
/*     45                    \! <EMPTY>                          */
   FILE$SEC$END = TRUE;
/*     46   <FILE-LIST> ::= <FILES>                              */
   ;     /* NO ACTION REQUIRED */
/*     47                  \! <FILE-LIST> <FILES>                */
   ;     /* NO ACTION REQUIRED */
/*     48   <FILES> ::= FD <ID> <FILE-CONTROL> .                 */
/*     48               <RECORD-DESCRIPTION>                     */
   DO;
     DO WHILE STACK$LEVEL > 1;
        CALL REDUCE$STACK;
     END;
     CALL END$OF$RECORD;
     REDEF = FALSE;
   END;
/*     49   <FILE-CONTROL> ::= <FILE-LIST>                       */
   CALL SET$IO$ADDRS;
/*     50                    \! <EMPTY>                          */
   CALL SET$IO$ADDRS;
/*     51   <FILE-LIST> ::= <FILE-ELEMENT>                       */
   ;     /* NO ACTION REQUIRED */
/*     52                   \! <FILE-LIST> <FILE-ELEMENT>        */
   ;     /* NO ACTION REQUIRED */
/*     53   <FILE-ELEMENT> ::= BLOCK <INTEGER> RECORDS           */
   ;     /* NO ACTION REQUIRED - FILES NEVER BLOCKED */
/*     54                    \! RECORD <REC-COUNT>               */
   CALL SET$SLENGTH(VALUE(SP));
/*     55                    \! LABEL RECORDS STANDARD           */
   ;     /* NO ACTION REQUIRED*/
/*     56                    \! LABEL RECORDS OMITTED            */
   ;     /* NO ACTION REQUIRED*/
/*     57                    \! VALUE OF <ID - STRING>           */
   ;     /* NO ACTION REQUIRED */
/*     58   <REC-COUNT> ::= <INTEGER>                            */
   ;     /* NO ACTION REQUIRED - VALUE(SP) CORRECT   */
/*     59                  \! <INTEGER> TO <INTEGER>             */
   DO;
   VALUE(MP) = VALUE(SP); /* VARIABLE LENGTH */
   CALL SET$TYPE(VARIABLE$LENG);   /* SET TO VARIABLE */
   END;
/*     60   <WORK> ::= WORKING-STORAGE SECTION .                 */
/*     60              <RECORD-DESCRIPTION>                      */
   DO;
     IF STACK$LEVEL<>77 THEN
       DO;
       DO WHILE STACK$LEVEL > 1;
         CUR$SYM = ID$STACK(ID$STACK$PTR - 1);
         IF GET$LEVEL = STACK$LEVEL THEN
           CALL REDEF$TEST;
         ELSE IF STACK$LEVEL > 1 THEN
```

```
                CALL REDUCE$STACK;
        END;
        IF STACK$LEVEL = 1 AND ID$STACK$PTR <> 0 THEN
          DO;
          CUR$SYM = ID$STACK(ID$STACK$PTR - 1);
          IF REDEF THEN CALL REDEF$TEST;
          END;
        END;
      CALL END$OF$RECORD;
    END;
/*      61            \! <EMPTY>                                    */
    ;     /* NO ACTION REQUIRED */
/*      62    <LINK> ::= LINKAGE SECTION .                          */
/*      62               <RECORD-DESCRIPTION>                       */
    ;     /* NO ACTION REQUIRED */
/*      63            \! <EMPTY>                                    */
    ;     /* NO ACTION REQUIRED */
/*      64    <RECORD-DESCRIPTION> ::= <LEVEL-ENTRY>                */
    ;     /* NO ACTION REQUIRED */
/*      65                           \! <RECORD-DESCRIPTION>        */
/*      65                              <LEVEL-ENTRY>               */
    ;     /* NO ACTION REQUIRED*/
/*      66    <LEVEL-ENTRY> ::= <INTEGER> <DATA-ID>                 */
/*      66                      <REDEFINES> <DATA-TYPE> .           */
    DO;
         CALL LOAD$LEVEL;
         IF (PENDING$LITERAL <> 0) AND (NOT VALUE$FLAG) THEN
            PENDING$LIT$ID = ID$STACK$PTR;
    END;
/*      67    <DATA-ID> ::= <ID>                                    */
    IF NOT UI$FLAG THEN
      DO;
        IF GET$TYPE = REL$KEY$UR THEN
          CALL SET$TYPE(REL$KEY);
        ELSE
          CALL PRINT$ERROR('DD');
      END;
/*      68               \! FILLER                                 */
    DO;
         CUR$SYM, VALUE(SP) = NEXT$SYM;
         CALL BUILD$SYMBOL(0);
    END;
/*      69    <REDEFINES> ::= REDEFINES <ID>                        */
    DO;
         IF UI$FLAG THEN
           CALL PRINT$ERROR('UD');
         CALL SET$REDEF(VALUE(SP),VALUE(SP - 2));
         VALUE(MP) = 1;     /* SET REDEFINE FLAG ON */
         IF NOT FILE$SEC$END THEN
           CALL PRINT$ERROR('R3');
         CALL CHECK$LVL$WORK;
```

```
   END;
/*     70                  \! <EMPTY>                                  */
   DO;
   IF NOT FILE$SEC$END THEN
     CALL CHECK$LVL$FILES;
   ELSE CALL CHECK$LVL$WORK;
   END;
/*     71    <DATA-TYPE> ::= <PROP-LIST>                               */
   ;     /* NO ACTION REQUIRED */
/*     72                  \! <EMPTY>                                  */
   ;     /* NO ACTION REQUIRED */
/*     73    <PROP-LIST> ::= <DATA-ELEMENT>                            */
   ;     /* NO ACTION REQUIRED */
/*     74                  \! <PROP-LIST> <DATA-ELEMENT>               */
   ;     /* NO ACTION REQUIRED */
/*     75    <DATA-ELEMENT> ::= PIC <INPUT>                            */
   CALL PIC$ANALIZER;
/*     76                     \! USAGE COMP                            */
   ;     /* NO ACTION REQUIRED-NOT IMPLEMENTED */
/*     77                     \! USAGE COMP-3                          */
   CALL SET$TYPE(COMP);
/*     78                     \! USAGE COMPUTATIONAL                   */
   ;     /* NO ACTION REQUIRED-NOTIMPLEMENTED */
/*     79                     \! USAGE DISPLAY                         */
   ;     /* NO ACTION REQUIRED - DEFAULT */
/*     80                     \! SIGN LEADING <SEPARATE>               */
   CALL SET$SIGN(17);
/*     81                     \! SIGN TRAILING <SEPARATE>              */
   CALL SET$SIGN(18);
/*     82                     \! OCCURS <INTEGER> INDEXED              */
/*     82                        <ID>                                  */
   ;     /* NO ACTION ACTION REQUIRED-NOT IMPLEMENTED */
/*     83                     \! OCCURS <INTEGER>                      */
   DO;
        CALL SET$TBL$SIZE(VALUE(SP));
        D$CNT = D$CNT + 1;
        CALL PROCESS$OCCURS;
        OCCURS$PTR = CUR$SYM;
        IF (TEMP$TWO := GET$LEVEL)=1 OR TEMP$TWO=77 THEN
          CALL PRINT$ERROR('OL');
   END;
/*     84                     \! SYNC <DIRECTION>                      */
   ;     /* NO ACTION REQUIRED - BYTE MACHINE */
/*     85                     \! VALUE <LITERAL>                       */
   IF NOT FILE$SEC$END THEN
     DO;
       CALL PRINT$ERROR('VE');
       PENDING$LITERAL = 0;
     END;
/*     86    <DIRECTION> ::= LEFT                                      */
   ;     /* NO ACTION REQUIRED */
```

```
/*     87                  \! RIGHT                                  */
   ;    /* NO ACTION REQUIRED */
/*     88                  \! <EMPTY>                                */
   ;    /* NO ACTION REQUIRED */
/*     89   <SEPARATE> ::= SEPARATE                                  */
   VALUE(SP) = 2;
/*     90                \! <EMPTY>                                  */
   ;    /* NO ACTION REQUIRED */
/*     91   <LITERAL> ::= <INPUT>                                    */
   DO;
       IF ((LIT$TYPE := GET$TYPE) < 16) OR
           (LIT$TYPE > 21) THEN
           DO;
               CALL PRINT$ERROR('NV');
               CALL LOAD$LITERAL(0);
               PENDING$LITERAL = 2;
           END;
       ELSE DO;
                CALL LOAD$LITERAL(1);
                PENDING$LITERAL = 1;
            END;
   END;
/*     92              \! <LIT>                                      */
   DO;
       CALL LOAD$LITERAL(0);
       PENDING$LITERAL = 2;
   END;
/*     93                \! ZERO                                     */
   PENDING$LITERAL = 3;
/*     94                \! SPACE                                    */
   PENDING$LITERAL = 4;
/*     95                 \! QUOTE                                   */
   PENDING$LITERAL = 5;
/*     96   <INTEGER> ::= <INPUT>                                    */
   CALL CONVERT$INTEGER;
/*     97   <ID> ::= <INPUT>                                         */
   DO;
       VALUE(SP) = MATCH; /* STORE SYMBOL TABLE POINTERS */
       IF FILE$DESC$FLAG THEN
           DO;
               FILE$DESC$FLAG = FALSE;
               IF UI$FLAG THEN
                   CALL PRINT$ERROR('UD');
               ELSE
                   IF GET$TYPE>UR$MASK THEN
                       CALL SET$TYPE(GET$TYPE - UR$MASK);
                   ELSE
                       CALL PRINT$ERROR('DD');
           END;
   END;
   END;   /* END OF CASE STATEMENT */
```

```
END CODE$GEN;

GETIN1: PROC BYTE;
    RETURN INDEX1(STATE);
END GETIN1;

GETIN2: PROC BYTE;
    RETURN INDEX2(STATE);
END GETIN2;

INCSP: PROC;
    IF (SP := SP + 1) >= PSTACKSIZE THEN
        CALL FATAL$ERROR('SO');
    VALUE(SP) = 0;    /* CLEAR VALUE STACK */
END INCSP;

LOOKAHEAD: PROC;
    IF NOLOOK THEN
        DO;
            CALL SCANNER;
            IF TOKEN = 2 THEN FILE$DESC$FLAG = TRUE;
            NOLOOK = FALSE;
            IF PRINT$TOKEN THEN
                DO;
                    CALL CRLF;
                    CALL PRINT$NUMBER(TOKEN);
                    CALL PRINT$CHAR(' ');
                    CALL PRINT$ACCUM;
                END;
        END;
END LOOKAHEAD;

NO$CONFLICT: PROC (CSTATE) BYTE;
    DCL (CSTATE,I,J,K) BYTE;
    J = INDEX1(CSTATE);
    K = J + INDEX2(CSTATE) - 1;
    DO I = J TO K;
        IF READ1(I) = TOKEN THEN RETURN TRUE;
    END;
RETURN FALSE;
END NO$CONFLICT;

RECOVER: PROC BYTE;
    DCL (TSP, RSTATE) BYTE;
    DO FOREVER;
        TSP = SP;
        DO WHILE TSP <> 255;
            IF NO$CONFLICT(RSTATE := STATESTACK(TSP)) THEN
                DO; /* STATE WILL READ TOKEN */
                    IF SP<>TSP THEN SP = TSP - 1;
                    RETURN RSTATE;
```

```
                END;
            TSP = TSP - 1;
        END;
        CALL SCANNER; /* TRY ANOTHER TOKEN */
    END;
END RECOVER;

END$PASS: PROC;
        /* THIS PROCEDURE STORES THE INFORMATION REQUIRED BY
        PASS2 IN LOCATIONS ABOVE THE SYMBOL TABLE. THE
        FOLLOWING INFORMATION IS STORED: INPUT BUFFER POINTER,
        OUTPUT FILE CONTROL BLOCK, COMPILER TOGGLES */
    CALL BYTE$OUT(SCD);
    CALL ADDR$OUT(NEXT$AVAILABLE);
    CALL MOVE(.DISPLAY(1),.LINE$CTR(0),5);
    OUTPUT$PTR = OUTPUT$PTR - .OUTPUT$BUFF;
    LIST$PTR = LIST$PTR - .LIST$BUFF;
    CALL MOVE(.DEBUGGING,MAX$MEMORY - PASS1$LEN,PASS1$LEN);
    L:  GO TO L;    /*  PATCH TO "JMP  0B000H"  */
END END$PASS;

 /*  *  *  *  *  PROGRAM EXECUTION STARTS HERE *  *  *  *  */

CALL MOVE(INITIAL$POS,MAX$MEMORY,RDR$LENGTH);
CALL INIT$SCANNER;
CALL INIT$SYMBOL;

    /* *  *  *  *  *  *  PARSER *  *  *  *  *  *  */

DO WHILE COMPILING;
    IF STATE <= MAXRNO THEN        /* READ STATE */
        DO;
            CALL INCSP;
            STATESTACK(SP) = STATE;  /* SAVE CURRENT STATE */
            CALL LOOKAHEAD;
            I = GETIN1;
            J = I + GETIN2 - 1;
            DO I = I TO J;
                IF READ1(I) = TOKEN THEN
                    DO;
                        /* COPY THE ACCUMULATOR IF IT IS AN
                        INPUT STRING.  IF IT IS A RESERVED
                        WORD IT DOES NOT NEED TO BE COPIED.*/
                            IF (TOKEN = INPUT$STR)
                                OR (TOKEN = LITERAL) THEN
                                DO K = 0 TO ACCUM(0);
                                    VARC(K) = ACCUM(K);
                                END;
                            STATE = READ2(I);
                            NOLOOK = TRUE;
                            I = J;
```

```
                    END;
                ELSE IF I = J THEN
                    DO;
                        CALL PRINT$ERROR('NP');
                        CALL PRINT(.(' ERROR NEAR $'));
                        CALL PRINT$ACCUM;
                        IF (STATE := RECOVER) = 0 THEN
                            COMPILING = FALSE;
                    END;
            END; /* DO I = I TO J; */
        END; /* END OF READ STATE */
    ELSE IF STATE > MAXPNO THEN /* APPLY PRODUCTION STATE */
        DO;
            MP = SP - GETIN2;
            MPP1 = MP + 1;
            CALL CODE$GEN(STATE - MAXPNO);
            SP = MP;
            I = GETIN1;
            J = STATESTACK(SP);
            DO WHILE (K := APPLY1(I)) <> 0 AND J <> K;
                I = I + 1;
            END;
            IF (K := APPLY2(I)) = 0 THEN COMPILING = FALSE;
            STATE = K;
        END;
    ELSE IF STATE <= MAXLNO THEN /*LOOKAHEAD STATE*/
        DO;
            I = GETIN1;
            CALL LOOKAHEAD;
            DO WHILE (K := LOOK1(I)) <> 0 AND TOKEN <> K;
                I = I + 1;
            END;
            STATE = LOOK2(I);
        END;
    ELSE
    DO; /*PUSH STATES*/
         CALL INCSP;
         STATESTACK(SP) = GETIN2;
         STATE = GETIN1;
    END;
END; /* DO WHILE COMPILING */
CALL END$PASS;
END;
```

```
$ TITLE('NPS MICRO-COBOL COMPILER PART 2') PAGEWIDTH(80)
    PAGEWIDTH(60)
PART2: DO;  /* MODULE NAME */

           /*     COBOL COMPILER - PART 2     */

           /*     MODULE LOCATED AT 103H      */

      /*     GLOBAL DECLARATIONS AND LITERALS     */

DECLARE DCL              LITERALLY 'DECLARE',
        LIT              LITERALLY 'LITERALLY';
DCL     FALSE            LIT       '0',
        ALPHA$LIT$FLAG   BYTE      INITIAL(FALSE),
        CR               LIT       '13',
        ERROR            BYTE      INITIAL(FALSE),
        FOREVER          LIT       'WHILE TRUE',
        IF$FLAG          BYTE      INITIAL(FALSE),
        LF               LIT       '10',
        MAX$MEMORY       ADDRESS   INITIAL(0B100H),
        PASS1$LEN        ADDRESS   INITIAL(353),
        PASS1$TOP        ADDRESS   INITIAL(0B000H),
        POUND            LIT       '23H',
        PROC             LIT       'PROCEDURE',
        QUOTE            LIT       '27H',
        TRUE             LIT       '1';

DCL     MAXLNO LIT '179', /* MAX LOOK COUNT */
        MAXPNO LIT '196', /* MAX PUSH COUNT */
        MAXRNO LIT '136', /* MAX READ COUNT */
        MAXSNO LIT '345', /* MAX STATE COUNT */
        PRODNO LIT '149', /* NUMBER OF PRODUCTIONS */
        STARTS LIT '1',   /* START STATE */
        ENDC   LIT '22',  /* END */
        FOFC   LIT '19',  /* EOF */
        PROCC  LIT '80',  /* PROCEDURE */
        TERMNO LIT '81';  /* TERMINAL COUNT */
DCL READ1(*) BYTE
    DATA(0,80,14,15,20,26,28,32,34,36,38,44,45,54,55,57,58,64
    ,65,69,70,75,77,63,3,41,63,63,3,4,7,41,63,78,41,63,42,41
    ,42,49,50,63,76,23,48,61,47,25,41,42,49,50,63,16,1,53,35
    ,63,74,1,72,3,43,56,39,2,10,11,31,46,66,68,81,14,15,20,26
    ,28,32,33,34,36,38,44,54,55,57,58,64,65,69,70,75,77,13,13
    ,30,13,51,5,8,41,52,63,73,78,21,6,21,11,71,60,60,71,60,71
    ,1,27,59,59,18,24,18,41,60,63,12,22,67,14,20,26,28,32,34
    ,38,44,54,55,57,58,64,65,69,70,75,77,29,41,60,63,29,67,1
    ,1,14,15,20,26,28,32,34,36,38,44,54,55,57,58,64,65,69,70
```

```
    ,75,77,4,7,4,6,7,14,15,17,20,26,28,32,33,34,36,38,44,54
    ,55,57,58,64,65,69,70,75,77,17,63,79,52,19,63,37,40,41,42
    ,49,50,63,6,9,3,41,42,49,50,63,0,0);
DCL LOOK1 (*) BYTE
    DATA(0,19,63,0,63,0,3,0,53,0,63,79,0,63,0,43,56,0,3,0,39
    ,0,5,8,0,5,8,0,5,8,0,5,8,0,5,8,0,41,52,63,73,0,21,0,21,0,
    ,71,0,71,0,60,71,0,60,71,0,71,0,71,0,71,0,71,0,71,0,2,10
    ,11,24,31,46,66,68,81,0,23,48,61,0,12,0,12,0,12,0,53,0,67
    ,0,63,0,63,0,27,59,0,4,7,0,63,0,17,0,63,0,37,0,40,41,42
    ,49,50,63,0,19,63,0);
DCL APPLY1 (*) BYTE
    DATA(0,0,113,0,19,0,0,128,0,0,134,0,71,105,110,119,123
    ,130,0,0,0,0,133,0,0,127,0,0,0,0,0,71,119,123,0,71,0,0
    ,105,110,130,0,0,0,6,0,7,8,10,11,0,9,12,0,15,0,105,110
    ,130,0,41,0,4,21,0,25,0,0,0,0,88,90,91,92,93,94,95,96,0,0
    ,0,0,0,0,114,0,0,0,0,0,102,0,16,17,22,23,28,30,47,48,49
    ,50,51,52,57,66,0,0,2,16,17,19,22,23,27,28,30,34,37,39,40
    ,42,43,44,45,47,48,49,50,51,52,54,55,57,62,66,115,116,122
    ,125,126,128,132,133,0,6,7,8,9,10,11,12,14,15,18,24,29,46
    ,59,60,81,103,111,0,16,17,22,23,28,30,44,47,48,49,50,51
    ,52,57,66,0,0,0,36,0,0,31,53,104,131,0,0,0,0);
DCL READ2 (*) ADDRESS
    DATA(0,63,19,345,24,26,138,31,33,34,36,39,40,43,44,45,46
    ,52,53,54,55,59,60,331,6,329,139,332,6,7,10,329,139,218
    329,139,333,329,334,336,335,139,249,322,320,321,313,301
    339,334,336,335,338,20,206,42,319,325,140,137,56,5,317
    ,319,37,296,295,297,293,294,292,287,288,19,345,24,26,138
    ,31,32,33,34,36,39,43,44,45,46,52,53,54,55,59,60,18,16,30
    ,17,234,9,12,329,41,139,57,61,25,286,25,14,298,49,50,298
    ,51,298,2,250,247,246,23,290,22,329,47,139,15,303,312,19
    24,26,138,31,33,36,39,43,44,45,46,52,53,54,55,59,60,28
    ,329,48,139,29,312,207,208,19,345,24,26,138,31,33,34,36
    ,39,43,44,45,46,52,53,54,55,59,60,8,11,8,276,11,19,345,21
    ,24,26,138,31,32,33,34,36,39,43,44,45,46,52,53,54,55,59
    ,60,21,326,62,41,197,326,35,38,329,334,336,335,139,330,13
    ,4,329,334,336,335,139,0,0);
DCL LOOK2 (*) ADDRESS
    DATA(0,204,204,3,27,180,326,327,58,181,200,200,220,66,182
    ,67,67,183,68,324,69,184,76,76,265,77,77,268,78,78,269,79
    ,79,266,80,80,267,81,81,81,81,185,83,280,85,281,87,186,88
    ,187,90,90,188,91,91,189,92,190,93,191,94,192,95,193,96
    ,194,195,195,195,101,195,195,195,195,195,284,102,102,102
    ,223,106,270,107,271,108,272,113,196,114,216,115,230,116
    ,231,248,248,119,120,120,260,122,215,124,238,125,198,129
    ,213,131,131,131,131,131,131,217,205,205,134);
DCL APPLY2 (*) ADDRESS
    DATA(0,0,214,97,126,176,128,203,202,179,118,117,306,244
    ,245,307,306,243,209,174,178,164,171,170,224,236,235,112
    ,127,72,240,308,309,308,210,99,98,71,213,213,213,177,103
    ,111,121,173,147,149,148,150,146,166,167,165,274,273,216
    ,216,216,169,175,123,84,153,152,283,282,285,70,104,252
```

```
     ,253,256,258,259,254,255,257,251,225,110,211,172,151,105
     ,302,130,279,82,314,133,132,89,157,154,222,158,156,155
     ,159,160,161,162,86,239,304,212,318,64,144,144,141,144
     ,144,342,144,144,344,310,226,142,200,143,144,277,144,144
     ,144,144,144,144,145,278,144,221,144,233,233,227,201,201
     ,64,232,232,65,275,275,275,275,275,275,275,275,275,242
     ,261,241,73,74,263,75,262,264,340,323,323,323,323,323,323
     ,323,323,323,323,323,323,323,323,323,323,328,135,168,337,341
     ,300,100,228,289,229,163,219,109,136);
DCL  INDEX1 (*) ADDRESS
     DATA(0,1,203,2,217,23,28,24,24,24,24,24,24,27,28,24,203
     ,203,34,203,33,217,203,203,34,217,36,203,203,34,203,37,42
     ,43,203,46,47,203,53,203,203,217,203,203,203,203,34,203
     ,203,203,203,203,203,37,203,203,54,203,55,34,34,56,203,58
     ,59,61,203,62,61,64,65,73,94,95,97,98,99,99,99,99,99,101
     ,105,106,107,106,109,110,110,111,112,114,110,110,110,110
     ,110,116,117,119,120,121,43,122,37,129,126,126,126,127
     ,129,147,151,55,152,203,203,153,154,155,175,177,203,180
     ,202,203,203,205,206,208,129,209,203,203,2,215,217,1.4,6
     ,8,10,13,15,18,20,22,25,28,31,34,37,42,44,46,48,50,53,56
     ,58,60,62,64,66,76,80,82,84,86,88,90,92,94,97,100,102,104
     ,,106,108,115,343,199,305,315,311,237,299,299,299,299,299
     ,299,299,299,299,291,199,1,2,2,4,4,6,6,7,7,7,9,9,10,10,10
     ,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,19,19,20,20,21
     ,21,22,22,24,24,24,24,25,27,28,29,30,31,31,31,31,31,35,35
     ,37,38,38,38,38,38,38,38,38,38,38,42,42,43,43,44,44,44,44
     ,44,46,46,46,51,51,54,54,56,56,56,60,60,62,62,65,65,65,67
     ,67,67,68,68,69,69,69,69,69,69,70,70,79,79,80,80,81,81,82
     ,82,82,82,83,83,84,86,87,87,88,88,89,89,90,90,90,92,92
     ,107,108,145,145,145,164,180,180,181,182,182,182,184,184
     ,184,185,185,190,190,191,192);
DCL  INDEX2 (*) BYTE
     DATA(0,1,1,21,6,1,5,3,3,3,3,3,3,1,5,3,1,1,2,1,1,6,1,1,2,6
     ,1,1,1,2,1,5,1,3,1,1,6,1,1,1,1,6,1,1,1,1,2,1,1,1,1,1,1,5
     ,1,1,1,1,1,2,2,2,1,1,2,1,1,2,1,1,8,21,1,2,1,1,2,2,2,2,2,4
     ,1,1,2,1,1,1,1,1,2,2,1,1,1,1,1,1,2,1,1,1,3,4,5,18,1,1,1,2
     ,18,4,1,1,1,1,1,1,1,20,2,3,1,22,1,1,2,1,2,1,18,6,1,1,21,2
     ,6,3,2,2,2,3,2,3,2,2,3,3,3,3,3,5,2,2,2,2,3,3,2,2,2,2,2,10
     ,4,2,2,2,2,2,2,2,2,3,3,2,2,2,2,2,7,3,27,58,66,67,69,81,87,88
     ,90,91,92,93,94,95,96,101,113,5,1,0,0,1,0,1,1,2,2,1,2,0,0
     ,2,1,0,2,1,0,1,4,1,1,3,3,1,3,1,0,1,0,1,1,2,0,1,1,0,2,0,1
     ,1,1,1,1,2,2,2,5,3,0,1,0,4,4,4,6,6,4,6,4,4,3,0,1,0,1,0,2
     ,2,1,1,0,2,2,0,2,0,2,1,1,2,0,2,0,2,0,2,2,0,0,1,0,0,0,0,0
     ,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,1,1,0,0,1,2,0,0,0,0,0,0,0
     ,0,0,0,0,0,0,0,0,3,0,2,0,0,0,0,0,0,0,0,0,0,0,1,0);

     /* END OF TABLES */
DECLARE
     /* JOINT DECLARATIONS */
     /* THE FOLLOWING ITEMS ARE DECLARED TOGETHER IN THIS
     GROUP IN ORDER TO FACILITATE THEIR BEING PASSED FROM
```

```
    THE FIRST PART OF THE COMPILER.
    */
    DEBUGGING           BYTE,
    ERROR$CTR(5)        BYTE,
    LINE$CTR(5)         BYTE,
    LIST$BUFF(128)      BYTE,
    LIST$FCB(33)        BYTE,
    LIST$INPUT          BYTE,
    LIST$PTR            ADDRESS,
    MAX$INT$MEM         ADDRESS,
    NEXT$AVAILABLE      ADDRESS,
    NEXT$SYM            ADDRESS,
    NO$CODE             BYTE,
    OUTPUT$BUFF(128)    BYTE,
    OUTPUT$FCB(33)      BYTE,
    OUTPUT$PTR          ADDRESS,
    POINTER             ADDRESS,
    PRINT$PROD          BYTE,
    PRINT$TOKEN         BYTE,
    SEQ$NUM             BYTE,
    WRITE$LST           BYTE,
    HASH$TAB$ADDR       ADDRESS, /* ADDRESS OF THE BOTTOM OF
                                    THE TABLES  FROM PART1 */

    /*  I O BUFFERS AND GLOBALS */

    IN$ADDR             ADDRESS INITIAL (5CH),
    INPUTFCB            BASED INADDR (33) BYTE,
    LIST$CHAR           BASED LIST$PTR BYTE,
    LIST$END            ADDRESS,
    OUTPUT$CHAR         BASED OUTPUT$PTR BYTE,
    OUTPUT$END          ADDRESS;

    /* GLOBAL PROCEDURES */

DECLARE
    CTR BYTE,
    A$CTR ADDRESS;

MON1: PROC (F,A) EXTERNAL;
    DCL F BYTE, A ADDRESS;
END MON1;

MON2: PROC (F,A) BYTE EXTERNAL;
    DCL F BYTE, A ADDRESS;
END MON2;

BOOT: PROC EXTERNAL;
      END BOOT;

PRINT$CHAR: PROC (CHAR);
```

```
    DCL CHAR BYTE;
    CALL MON1 (2,CHAR);
END PRINTCHAR;

WRITE$OUTPUT: PROC (BUFF,FCB);
    DCL (BUFF,FCB) ADDRESS;
    CALL MON1(26,BUFF); /* SET DMA */
    IF MON2(21,FCB) <> 0 THEN
        DO;
            CALL MON1(9,.('WR$'));
            CALL BOOT;
        END;
    CALL MON1(26,80H);  /*RESET DMA */
END WRITE$OUTPUT;

WRITE$TO$DISK: PROC(CHAR);
    DCL CHAR BYTE;
    IF (LIST$PTR := LIST$PTR + 1) > LIST$END THEN
        DO;
            CALL WRITE$OUTPUT(.LIST$BUFF,.LIST$FCB);
            LIST$PTR = .LIST$BUFF;
        END;
    LIST$CHAR = CHAR;
END WRITE$TO$DISK;

PRINT: PROC (A);
    DCL (A,ADDR) ADDRESS,CHAR BASED ADDR BYTE;
    ADDR = A;
    CALL MON1 (9,A);
    DO WHILE CHAR <> '$';
        CALL WRITE$TO$DISK(CHAR);
        ADDR = ADDR + 1;
    END;
END PRINT;

CRLF: PROC;
    CALL MON1(9,.(CR,LF,'$'));
END CRLF;

DCRLF: PROC;
    CALL WRITE$TO$DISK(CR);
    CALL WRITE$TO$DISK(LF);
END DCRLF;

INC$CTR: PROC(BASE);
    DCL BASE ADDRESS, CTR BYTE, B$BYTE BASED BASE (1) BYTE,
        TEN LIT '3AH';
    CTR = 4;
    DO WHILE (B$BYTE(CTR) := B$BYTE(CTR) + 1) = TEN;
        B$BYTE(CTR) = '0';
        IF CTR > 0 THEN
```

```
            IF B$BYTE(CTR := CTR - 1) = ' ' THEN
                B$BYTE(CTR) = '0';
    END;
END INC$CTR;

PRINT$ERROR: PROC (CODE);
    DCL CODE ADDRESS,CODE1(6) ADDRESS,I BYTE;
    IF CODE = FALSE THEN
        DO;
            DO I = 0 TO 5;
                CODE1(I) = 0;
            END;
            I = 0;
        END;
    ELSE IF CODE = TRUE THEN
        DO;
            I = 0;
            DO WHILE((I <> 6) AND (CODE1(I) <> 0));
                CALL PRINTCHAR(HIGH(CODE1(I)));
                CALL PRINTCHAR(LOW (CODE1(I)));
                CALL WRITE$TO$DISK(HIGH(CODE1(I)));
                CALL WRITE$TO$DISK(LOW (CODE1(I)));
                CALL CRLF;
                CALL DCRLF;
                CODE1(I) = 0;
                I = I + 1;
            END;
            I = 0;
            ERROR = FALSE;
        END;
    ELSE IF (CODE = 'NP') OR (CODE = 'NV')
         OR (CODE = 'SL') THEN
        DO;
            ERROR = TRUE;
            CALL PRINTCHAR(HIGH(CODE));
            CALL PRINTCHAR(LOW (CODE));
            CALL WRITE$TO$DISK(HIGH(CODE));
            CALL WRITE$TO$DISK(LOW (CODE));
            CALL INC$CTR(.ERROR$CTR(0));
            IF CODE <> 'NP' THEN
               DO;
                   CALL CRLF;
                   CALL DCRLF;
               END;
        END;
     ELSE DO;
              ERROR = TRUE;
              IF I <> 6 THEN
                  DO;
                      CODE1(I) = CODE;
                      I = I + 1;
```

```
                END;
            CALL INC$CTR(.ERROR$CTR(0));
        END;
END PRINT$ERROR;

FATAL$ERROR: PROC(REASON);
    DCL REASON ADDRESS;
    CALL PRINT$ERROR(REASON);
    CALL PRINT$ERROR(TRUE);
    CALL BOOT;
END FATAL$ERROR;

CLOSE: PROC(FCB);
    DCL FCB ADDRESS;
    IF MON2(16,FCB) = 255 THEN CALL FATAL$ERROR('CL');
END CLOSE;

MORE$INPUT: PROC BYTE;
    DCL DCNT BYTE;
    IF (DCNT := MON2(20,.INPUT$FCB)) > 1 THEN
        CALL FATAL$ERROR('BR');
    RETURN NOT(DCNT);
END MORE$INPUT;

MOVE: PROC(SOURCE, DESTINATION, COUNT);
    DCL (COUNT,SOURCE,DESTINATION) ADDRESS,
    (S$BYTE BASED SOURCE, D$BYTE BASED DESTINATION) BYTE;
    DO WHILE (COUNT := COUNT - 1) <> 0FFFFH;
        D$BYTE = S$BYTE;
        SOURCE = SOURCE +1;
        DESTINATION = DESTINATION + 1;
    END;
END MOVE;

FILL: PROC(ADDR,CHAR,COUNT);
    DCL (ADDR,COUNT) ADDRESS,
    (CHAR,DEST BASED ADDR) BYTE;
    DO WHILE (COUNT := COUNT - 1) <> 0FFFFH;
        DEST=CHAR;
        ADDR=ADDR + 1;
    END;
END FILL;

    /*   *  *  *  *  *  SCANNER LITS *  *  *  *  */

DECLARE
    INPUT$STR       LIT       '63',
    INVALID         LIT        '0',
    LITERAL         LIT       '42',
    LPARIN          LIT        '3',
    PERIOD          LIT        '1',
```

```
    RPARIN          LIT         '6';

    /*  * * * * SCANNER TABLES * * * *    */

DCL TOKEN$TABLE (*) BYTE DATA
    /* CONTAINS THE TOKEN NUMBER ONE LESS THAN THE FIRST
    RESERVED WORD FOR EACH LENGTH OF WORD  */
    (0,0,12,18,25,42,54,63,73,77,80),

TABLE (*) BYTE DATA('BY','GO','IF','NO','OR','TO','EOF','ADD'
    ,'AND','END','I-O','NOT','RUN','CALL','ELSE','EXIT'
    ,'FROM','INTO','LESS','MOVE','NEXT','OPEN','PAGE','READ'
    ,'SIZE','STOP','THRU','WITH','ZERO','AFTER','CLOSE'
    ,'ENTER','EQUAL','ERROR','INPUT','QUOTE','TIMES','SPACE'
    ,'UNTIL','USING','WRITE','ACCEPT','BEFORE','DELETE'
    ,'DIVIDE','END-IF','GIVING','OUTPUT','COMPUTE','DISPLAY'
    ,'GREATER','INVALID','NUMERIC','PERFORM','REWRITE'
    ,'ROUNDED','SECTION','VARYING','DIVISION','MULTIPLY'
    ,'SENTENCE','SUBTRACT','ADVANCING','DEPENDING'
   ,'PROCEDURE','ALPHABETIC'),
OFFSET (11) ADDRESS INITIAL
    /*  NUMBER OF BYTES TO INDEX INTO THE TABLE FOR EACH
        LENGTH */
    (0,0,0,12,33,97,157,199,269,301,328),

WORD$COUNT (*) BYTE DATA
    /* NUMBER OF WORDS OF EACH SIZE */
    (0,0,6,7,16,12,7,10,4,3,1),

    ACCUM(82)       BYTE,
    ADD$END(*)      BYTE    DATA('      EOF  '),
    BUFFER$END      ADDRESS INITIAL(100H),
    CHAR            BYTE    INITIAL(' '),
    DISPLAY(88)     BYTE    INITIAL(0),
    EOFFILLER       LIT     '1AH',
    FIRST$LINE      BYTE    INITIAL(TRUE),
    FORMFEED        LIT     '0CH',
    HOLD            BYTE,
    INBUFF          LIT     '80H',
    LOOKED          BYTE    INITIAL(0),
    MAX$ID$LEN      LIT      '15',
    MAX$LEN         LIT      '10',
    NEXT            BASED    POINTER  BYTE,
    TAB             LIT      '09',
    TOKEN           BYTE;   /*RETURNED FROM SCANNER */

    /*  PROCS USED BY THE SCANNER */

NEXT$CHAR: PROC BYTE;
    IF LOOKED THEN
        DO;
```

```
            LOOKED = FALSE;
            RETURN (CHAR := HOLD);
        END;
    IF (POINTER := POINTER + 1) >= BUFFER$END THEN
        DO;
            IF NOT MORE$INPUT THEN
                DO;
                    BUFFER$END = .MEMORY;
                    POINTER = .ADD$END;
                END;
            ELSE POINTER = INBUFF;
        END;
    IF NEXT = EOFFILLER THEN
        DO;
            BUFFER$END = .MEMORY;
            POINTER = .ADD$END;
        END;
    RETURN (CHAR := NEXT);
END NEXT$CHAR;

GET$CHAR: PROC;
    CHAR = NEXT$CHAR;
END GET$CHAR;

DISPLAY$LINE: PROC;
    DCL I BYTE;
    DO I = 1 TO DISPLAY(0);
        IF LIST$INPUT OR ERROR THEN
            CALL PRINT$CHAR(DISPLAY(I));
        IF WRITE$LST OR ERROR THEN
            CALL WRITE$TO$DISK(DISPLAY(I));
    END;
    IF FIRST$LINE THEN
        DO;
            CALL MOVE(.LINE$CTR,.DISPLAY(1),5);
            FIRST$LINE = FALSE;
        END;
    ELSE CALL INC$CTR(.DISPLAY(0));
    DISPLAY(0) = 5;
END DISPLAY$LINE;

LOAD$DISPLAY: PROC;
    IF DISPLAY(0) < 87 THEN
        DISPLAY(DISPLAY(0) := DISPLAY(0) + 1) = CHAR;
    CALL GET$CHAR;
END LOAD$DISPLAY;

PUT: PROC;
    IF ACCUM(0) < 81 THEN
    ACCUM(ACCUM(0) := ACCUM(0) + 1) = CHAR;
    CALL LOAD$DISPLAY;
```

```
END PUT;

EAT$LINE: PROC;
    DO WHILE CHAR <> CR;
        CALL LOAD$DISPLAY;
    END;
END EAT$LINE;

GET$NO$BLANK: PROC;
    DCL I BYTE;
    DO FOREVER;
        IF CHAR = ' ' OR CHAR = TAB THEN CALL LOAD$DISPLAY;
        ELSE IF CHAR=CR THEN
            DO;
                CALL LOAD$DISPLAY;
                CALL LOAD$DISPLAY;
                CALL DISPLAY$LINE;
                CALL PRINT$ERROR(TRUE);
                DO WHILE CHAR = CR;
                    CALL LOAD$DISPLAY;
                    CALL LOAD$DISPLAY;
                    CALL DISPLAY$LINE;
                END;
                IF SEQ$NUM THEN
                    DO I = 1 TO 6;
                        CALL LOAD$DISPLAY;
                    END;
                IF CHAR = '*' THEN CALL EAT$LINE;
                ELSE IF CHAR = '/' THEN
                    DO;
                        IF LIST$INPUT THEN
                            CALL PRINT$CHAR(FORM$FEED);
                        IF WRITE$LST THEN
                            CALL WRITE$TO$DISK(FORM$FEED);
                        CALL EAT$LINE;
                    END;
                ELSE IF CHAR = ':' THEN
                    IF NOT DEBUGGING THEN CALL EAT$LINE;
                    ELSE CALL LOAD$DISPLAY;
            END;
        ELSE RETURN;
    END;    /* END OF DO FOREVER */
END GET$NO$BLANK;

SPACE: PROC BYTE;
    RETURN (CHAR = ' ') OR (CHAR = CR) OR (CHAR = TAB);
END SPACE;

LEFT$PARIN: PROC BYTE;
    RETURN CHAR = '(';
END LEFT$PARIN;
```

```
RIGHT$PARIN: PROC BYTE;
    RETURN CHAR = ')';
END RIGHT$PARIN;

DELIMITER: PROC BYTE;
    IF CHAR <> '.' THEN RETURN FALSE;
    HOLD = NEXT$CHAR;
    LOOKED = TRUE;
    IF SPACE THEN
        DO;
             CHAR = '.';
             RETURN TRUE;
        END;
    CHAR = '.';
    RETURN FALSE;
END DELIMITER;

END$OF$TOKEN: PROC BYTE;
    RETURN SPACE OR  DELIMITER OR LEFT$PARIN OR RIGHT$PARIN;
END END$OF$TOKEN;

GET$LITERAL: PROC BYTE;
    CALL LOAD$DISPLAY;
    DO FOREVER;
        IF CHAR = QUOTE THEN
            DO;
              CALL LOAD$DISPLAY;
              RETURN LITERAL;
            END;
        CALL PUT;
    END;
END GET$LITERAL;

LOOK$UP: PROC BYTE;
    DCL POINT ADDRESS,
    HERE BASED POINT (1) BYTE, I BYTE;

    MATCH: PROC BYTE;
        DCL J BYTE;
        DO J = 1 TO ACCUM(0);
             IF HERE(J - 1) <> ACCUM(J) THEN RETURN FALSE;
        END;
        RETURN TRUE;
    END MATCH;

    POINT = OFFSET(ACCUM(0)) + .TABLE;
    DO I = 1 TO WORD$COUNT(ACCUM(0));
        IF MATCH THEN RETURN I;
        POINT = POINT + ACCUM(0);
    END;
```

```
    RETURN FALSE;
END LOOK$UP;

RESERVED$WORD: PROC BYTE;
    DCL (NUMB,VALUE) BYTE;
    IF ACCUM(Ø) <= MAX$LEN THEN
        DO;
            IF (NUMB := TOKEN$TABLE(ACCUM(Ø))) <> Ø THEN
                IF (VALUE := LOOK$UP) <> Ø THEN
                    NUMB = NUMB + VALUE;
                ELSE NUMB = Ø;
        END;
    ELSE NUMB = Ø;
    RETURN NUMB;
END RESERVED$WORD;

GET$TOKEN: PROC BYTE;
    ACCUM(Ø) = Ø;
    CALL GET$NO$BLANK;
    IF CHAR = QUOTE THEN RETURN GET$LITERAL;
    IF DELIMITER THEN
        DO;
            CALL PUT;
            RETURN PERIOD;
        END;
    IF LEFT$PARIN THEN
        DO;
            CALL PUT;
            RETURN LPARIN;
        END;
    IF RIGHT$PARIN THEN
        DO;
            CALL PUT;
            RETURN RPARIN;
        END;
    DO FOREVER;
        CALL PUT;
        IF END$OF$TOKEN THEN RETURN INPUT$STR;
    END; /* OF DO FOREVER */
END GET$TOKEN;
    /*    END OF SCANNER ROUTINES  */
    /*    SCANNER EXEC  */
SCANNER: PROC;
    IF(TOKEN := GET$TOKEN) = INPUT$STR THEN
        IF (CTR := RESERVED$WORD) <> Ø THEN TOKEN = CTR;
END SCANNER;

PRINT$ACCUM: PROC;
    DCL I BYTE;
    DO I = 1 TO ACCUM(Ø);
        CALL PRINT$CHAR(ACCUM(I));
```

```
        CALL WRITE$TO$DISK(ACCUM(I));
    END;
    CALL CRLF;
    CALL DCRLF;
END PRINT$ACCUM;

PRINT$NUMBER: PROC(NUMB);
    DECLARE(NUMB,I,CNT,K) BYTE, J (*) BYTE DATA(100,10);
    DO I = 0 TO 1;
        CNT = 0;
        DO WHILE NUMB >= (K := J(I));
            NUMB = NUMB - K;
            CNT = CNT + 1;
        END;
        CALL PRINTCHAR('0' + CNT);
    END;
    CALL PRINTCHAR('0' + NUMB);
END PRINT$NUMBER;
    /*  *  *  *  END OF SCANNER PROCS  *  *  *  */

    /*  *  *  *  *  SYMBOL TABLE DECLARATIONS * * *  */
DECLARE
CUR$SYM             ADDRESS,       /*SYMBOL BEING ACCESSED*/
DECIMAL             LIT             '11',
DISPLACEMENT        LIT             '14',
FCB$ADDR            LIT              '4',
FLD$LENGTH          LIT              '3',
HASH$MASK           LIT            '3FH',
LEVEL               LIT             '10',
LOCATION            LIT              '2',
P$LENGTH            LIT              '3',
REL$ID              LIT              '5',
S$TYPE              LIT              '2',
START$NAME          LIT             '13', /*1 LESS*/
SYMBOL              BASED CUR$SYM (1)  BYTE,
SYMBOL$ADDR         BASED CUR$SYM (1)  ADDRESS,
TEMP$PTR            ADDRESS,
TEMP$ADDR           BASED TEMP$PTR ADDRESS,

     /* * * * * * * SYMBOL TYPE LITERALS * * * * * * * */

A$ED                LIT             '72',
A$N$ED              LIT             '73',
ALPHA               LIT              '8',
ALPHA$NUM           LIT              '9',
COMP                LIT             '21',
GROUP               LIT              '6',
LABEL$TYPE          LIT             '32',
LIT$QUOTE           LIT             '11',
LIT$SPACE           LIT             '10',
LIT$ZERO            LIT             '12',
```

```
MULT$OCCURS         LIT             '128',
NON$NUMERIC$LIT     LIT               '7',
NUM$ED              LIT              '80',
NUMERIC             LIT              '16',
NUMERIC$LITERAL     LIT              '15',
UNRESOLVED          LIT             '255';

    /*  *  *  *  SYMBOL TABLE ROUTINES  *  *  *  */

SET$ADDRESS: PROC(ADDR);
DCL ADDR ADDRESS;
    SYMBOL$ADDR(LOCATION) = ADDR;
END SET$ADDRESS;

GET$ADDRESS: PROC ADDRESS;
    RETURN SYMBOL$ADDR(LOCATION);
END GET$ADDRESS;

GET$FCB$ADDR: PROC ADDRESS;
    RETURN SYMBOL$ADDR(FCB$ADDR);
END GET$FCB$ADDR;

GET$TYPE: PROC BYTE;
    RETURN SYMBOL(S$TYPE);
END GET$TYPE;

SET$TYPE: PROC(TYPE);
    DCL TYPE BYTE;
    SYMBOL(S$TYPE) = TYPE;
END SET$TYPE;

GET$LENGTH: PROC ADDRESS;
    RETURN SYMBOL$ADDR(FLD$LENGTH);
END GET$LENGTH;

GET$LEVEL: PROC BYTE;
    RETURN SYMBOL(LEVEL);
END GET$LEVEL;

GET$DECIMAL: PROC BYTE;
    RETURN SYMBOL(DECIMAL);
END GET$DECIMAL;

GET$P$LENGTH: PROC BYTE;
    RETURN SYMBOL(P$LENGTH);
END GET$P$LENGTH;

BUILD$SYMBOL: PROC(LEN);
    DCL LEN BYTE, TEMP ADDRESS;
    TEMP = NEXT$SYM;
    IF (NEXT$SYM := .SYMBOL(LEN := LEN + DISPLACEMENT))
```

```
        > MAX$MEMORY THEN CALL FATAL$ERROR('ST');
    CALL FILL (TEMP,0,LEN);
END BUILD$SYMBOL;

GET$PREV$OCCURS: PROC ADDRESS;
    TEMP$PTR = CUR$SYM + DISPLACEMENT + GET$P$LENGTH;
    RETURN TEMP$ADDR;
END GET$PREV$OCCURS;

AND$OUT$OCCURS: PROC (TYPE$IN) BYTE;
    DCL TYPE$IN BYTE;
    RETURN TYPE$IN  AND 127;
END AND$OUT$OCCURS;

CHECK$UNRESOLVED: PROC;
    DCL (I,J) BYTE,PTR ADDRESS,ADDR$PTR BASED PTR ADDRESS;
    PTR = HASH$TAB$ADDR;/*SET PTR TO FIRST HASH ADDR*/
    DO I = 1 TO 64;
      IF ADDR$PTR<>0 THEN
        DO;
          CUR$SYM = ADDR$PTR;
          DO WHILE CURSYM<>0;
            IF GET$TYPE = UNRESOLVED THEN
              DO;
                CALL PRINT(.('UL $'));
                DO J = 1 TO GET$P$LENGTH;
                  CALL PRINT$CHAR(SYMBOL(START$NAME + J));
                  CALL WRITE$TO$DISK(SYMBOL(START$NAME + J));
                END;
                CALL CRLF;
                CALL DCRLF;
                CALL INC$CTR(.ERROR$CTR(0));
              END;
            CURSYM = SYMBOL$ADDR(0);
          END;
        END;
      PTR = PTR + 2;
    END;
END CHECK$UNRESOLVED;

    /*  *  *  *  PARSER DECLARATIONS  *  *  *  */

DCL
COMPILING               BYTE      INITIAL(TRUE),
CON$LENGTH              BYTE,
COND$TYPE               BYTE,
DISPLAY$FLAG            BYTE      INITIAL(FALSE),
HOLD$SEC$ADDR           ADDRESS,
HOLD$SECTION            ADDRESS,
ID$PTR                  BYTE,
ID$STACK(20)            ADDRESS,
```

```
(I,J,K)                 ADDRESS,  /*INDICIES FOR THE PARSER*/
L$ADDR                  ADDRESS,
L$DEC                   BYTE,
L$DEC$TEMP              BYTE,
L$LENGTH                ADDRESS,
L$TYPE                  BYTE,
MP                      BYTE,
MPP1                    BYTE,
NEXT$ADDRESS            ADDRESS   INITIAL(0),
NOLOOK                  BYTE      INITIAL(FALSE),
PSTACKSIZE              LIT       '30', /* SIZE OF STACKS*/
SECTION$FLAG            BYTE      INITIAL(0),
SP                      BYTE      INITIAL(255),
STATE                   ADDRESS   INITIAL(STARTS),
STATESTACK(PSTACKSIZE)  ADDRESS,  /* SAVED STATES */
SUB$IND                 BYTE      INITIAL(0),
VARC(100)               BYTE,     /*TEMP CHAR STORE*/
VALUE(PSTACKSIZE)       ADDRESS,  /* TEMP VALUES */
VALUE2(PSTACKSIZE)      ADDRESS,  /* VALUE2  STACK */
WRITE$BEFORE            BYTE      INITIAL(FALSE),
WRITE$AFTER             BYTE      INITIAL(FALSE),

    /* * * * * * * * CODE LITERALS * * * * * * * * * * */
    /* THE CODE LITERALS ARE BROKEN INTO GROUPS DEPENDING
    ON THE TOTAL LENGTH OF CODE PRODUCED FOR THAT ACTION */
    /* LENGTH ONE */

ADD LIT  '1',   /* ADD  REGISTER 1 TO REGISTER 0 */
SUB LIT  '2',   /* SUBTRACT  REGISTER 1 FROM REGISTER 0 */
MUL LIT  '3',   /* MULTIPLY REGISTER 0 BY REGISTER 1 */
DIV LIT  '4',   /* DIVIDE REGISTER 0 BY REGISTER 1 */
NEG LIT  '5',   /* NOT OPERATOR */
STP LIT  '6',   /* STOP PROGRAM */
STI LIT  '7',   /* STORE REGISTER 2 INTO REGISTER 0 */
EXT LIT  '8',   /* EXIT SUBROUTINE */
      /* LENGTH TWO */
RND LIT  '9',   /* ROUND CONTENTS OF REGISTER 2 */
      /* LENGTH THREE */
RET LIT  '10',  /* RETURN */
CLS LIT  '11',  /* CLOSE */
SER LIT  '12',  /* BRANCH ON SIZE ERROR */
BRN LIT  '13',  /* BRANCH */
OPN LIT  '14',  /* OPEN A FILE FOR INPUT */
OP1 LIT  '15',  /* OPEN A FILE FOR OUTPUT */
OP2 LIT  '16',  /* OPEN A FILE FOR BOTH INPUT AND OUTPUT */
RGT LIT  '17',  /* REGISTER GREATER THAN */
RLT LIT  '18',  /* REGISTER LESS THAN */
REQ LIT  '19',  /* REGISTER EQUAL */
INV LIT  '20',  /* BRANCH IF INVALID-FILE-ACTION FLAG TRUE */
EOR LIT  '21',  /* BRANCH ON END-OF-RECORDS FLAG */
      /* LENGTH FOUR */
```

```
PAG LIT  '22',  /* CARRIAGE CONTROL FOR PRINTER OPERATION */
ACC LIT  '23',  /* ACCEPT */
STD LIT  '24',  /* STOP WITH DISPLAY */
LDI LIT  '25',  /* LOAD A CODE ADDRESS DIRECT */
      /* LENGTH FIVE */
DIS LIT  '26',  /* DISPLAY */
DEC LIT  '27',  /* DECREMENT COUNT AND BRANCH IF ZERO */
STO LIT  '28',  /* STORE NUMERIC */
ST1 LIT  '29',  /* STORE SIGNED NUMERIC LEADING */
ST2 LIT  '30',  /* STORE SIGNED NUMERIC TRAILING */
ST3 LIT  '31',  /* STORE SEPARATE SIGN LEADING */
ST4 LIT  '32',  /* STORE SEPARATE SIGN TRAILING */
ST5 LIT  '33',  /* STORE A PACKED NUMERIC FIELD */
      /* LENGTH SIX */
LOD LIT  '34',  /* LOAD NUMERIC LITERAL */
LD1 LIT  '35',  /* LOAD NUMERIC */
LD2 LIT  '36',  /* LOAD SIGNED NUMERIC LEADING */
LD3 LIT  '37',  /* LOAD SIGNED NUMERIC TRAILING */
LD4 LIT  '38',  /* LOAD SEPARATE SIGN LEADING */
LD5 LIT  '39',  /* LOAD SEPARATE SIGN TRAILING */
LD6 LIT  '40',  /* LOAD A PACKED NUMERIC FIELD */
      /* LENGTH SEVEN */
PER LIT  '41',  /* PERFORM */
CNU LIT  '42',  /* COMPARE NUMERIC UNSIGNED */
CNS LIT  '43',  /* COMPARE NUMERIC SIGNED */
CAL LIT  '44',  /* COMPARE  ALPHABETIC */
RWS LIT  '45',  /* REWRITE SEQUENTIAL */
DLS LIT  '46',  /* DELETE SEQUENTIAL */
RDF LIT  '47',  /* READ A SEQUENTIAL FILE */
WTF LIT  '48',  /* WRITE A RECORD TO A SEQUENTIAL FILE */
RVL LIT  '49',  /* READ A VARIABLE LENGTH FILE */
WVL LIT  '50',  /* WRITE A VARIABLE LENGTH RECORD */
      /* LENGTH NINE */
SCR LIT  '51',  /* CALCULATE A SUBSCRIPT */
SGT LIT  '52',  /* STRING GREATER THAN */
SLT LIT  '53',  /* STRING LESS TEAN */
SEQ LIT  '54',  /* STRING EQUAL */
MOV LIT  '55',  /* MOVE */
      /* LENGTH TEN */
RRS LIT  '56',  /* READ RELATIVE SEQUENTIAL */
WRS LIT  '57',  /* WRITE RELATIVE SEQUENTIAL */
RRR LIT  '58',  /* READ RELATIVE RANDOM */
WRR LIT  '59',  /* WRITE RELATIVE RANDOM */
RWR LIT  '60',  /* REWRITE RELATIVE */
DLR LIT  '61',  /* DELETE RELATIVE */
      /* LENGTH ELEVEN */
MED LIT  '62',  /* MOVE INTO AN ALPHANUMERIC EDITED FIELD */
      /* LENGTH THIRTEEN */
MNE LIT  '63',  /* MOVE INTO A NUMERIC EDITED FIELD */
SBR LIT  '64',  /* SUBROUTINE CALL */
      /* VARIABLE LENGTH */
```

```
GDP LIT  '65',  /* GO TO - DEPENDING ON */
PAR LIT  '66',  /* PARAMETER LIST */
      /* BUILD DIRECTING ONLY */
INT LIT  '67',  /* INITIALIZE MEMORY */
BST LIT  '68',  /* BACK STUFF  */
TER LIT  '69',  /* TERMINATE BUILD */
SCD LIT  '70';  /* START CODE */

    /*  *  *  *  PARSER ROUTINES  *  *  *  *  */
DIGIT: PROC (CHAR) BYTE;
    DCL CHAR BYTE;
    RETURN (CHAR <= '9') AND (CHAR >= '0');
END DIGIT;

LETTER: PROC (CHAR) BYTE;
    DCL CHAR BYTE;
    RETURN (CHAR >= 'A') AND (CHAR <= 'Z');
END LETTER;

INVALID$TYPE: PROC;
    CALL PRINT$ERROR('IT');
END INVALID$TYPE;

BYTE$OUT: PROC(ONE$BYTE);
    DCL ONE$BYTE BYTE;
    IF NO$CODE THEN RETURN;
    IF (OUTPUT$PTR := OUTPUT$PTR + 1) > OUTPUT$END THEN
    DO;
        CALL WRITE$OUTPUT(.OUTPUT$BUFF,.OUTPUT$FCB);
        OUTPUT$PTR = .OUTPUT$BUFF;
    END;
    OUTPUT$CHAR = ONE$BYTE;
END BYTE$OUT;

ADDR$OUT: PROC (ADDR);
    DCL ADDR ADDRESS;
    CALL BYTE$OUT(LOW(ADDR));
    CALL BYTE$OUT(HIGH (ADDR));
END ADDR$OUT;

INC$COUNT: PROC(CNT);
    DCL CNT BYTE;
    IF(NEXT$AVAILABLE := NEXT$AVAILABLE + CNT)
        > MAX$INT$MEM THEN CALL FATAL$ERROR('MO');
END INC$COUNT;

ONE$ADDR$OPP: PROC(CODE,ADDR);
    DCL CODE BYTE, ADDR ADDRESS;
    CALL BYTE$OUT(CODE);
    CALL ADDR$OUT(ADDR);
    CALL INC$COUNT(3);
```

```
END ONE$ADDR$OPP;

MATCH: PROC ADDRESS;
    DCL POINT ADDRESS, COLLISION BASED POINT ADDRESS,
        (HOLD,I) BYTE;
    IF VARC(Ø)>MAX$ID$LEN THEN VARC(Ø) = MAX$ID$LEN;
    HOLD = Ø;
    DO I = 1 TO VARC(Ø);
        HOLD = HOLD + VARC(I);
    END;
    POINT = HASH$TAB$ADDR + SHL((HOLD AND HASH$MASK),1);
    DO FOREVER;
        IF COLLISION = Ø THEN
            DO;
                CUR$SYM,COLLISION = NEXT$SYM;
                CALL BUILD$SYMBOL(VARC(Ø));
                SYMBOL(P$LENGTH) = VARC(Ø);
                DO I = 1 TO VARC(Ø);
                    SYMBOL(START$NAME + I) = VARC(I);
                END;
                CALL SET$TYPE(UNRESOLVED);
                RETURN CUR$SYM;
            END;
        ELSE
            DO;
              CUR$SYM=COLLISION;
              IF (HOLD:=GET$P$LENGTH)=VARC(Ø) THEN
                DO;
                    I=1;
                    DO WHILE SYMBOL(START$NAME + I)= VARC(I);
                     IF (I:=I+1)>HOLD THEN
                          RETURN(CUR$SYM := COLLISION);
                    END;
                END;
            END;
        POINT = COLLISION;
    END;
END MATCH;

SET$VALUE: PROC(NUMB);
    DCL NUMB ADDRESS;
    VALUE(MP) = NUMB;
END SET$VALUE;

SET$VALUE2: PROC(ADDR);
    DCL ADDR ADDRESS;
    VALUE2(MP) = ADDR;
END SET$VALUE2;

CHK$UD$VAR:PROC(PTR);
    DCL PTR BYTE;
```

```
    CURSYM = VALUE(PTR);
    IF GET$TYPE = UNRESOLVED THEN
      CALL PRINT$ERROR('UD');
END CHK$UD$VAR;

SUB$CNT: PROC BYTE;
    IF (SUB$IND := SUB$IND + 1) > 7 THEN
         SUB$IND = 1;
    RETURN SUB$IND;
END SUB$CNT;

CODE$BYTE: PROC (CODE);
    DCL CODE BYTE;
    CALL BYTE$OUT(CODE);
    CALL INC$COUNT(1);
END CODE$BYTE;

CODE$ADDRESS: PROC (CODE);
    DCL CODE ADDRESS;
    CALL ADDR$OUT(CODE);
    CALL INC$COUNT(2);
END CODE$ADDRESS;

CONVERT$INTEGER: PROC ADDRESS;
    DCL A BYTE;
    ACTR = 0;
    IF VARC(1) = '+' THEN A = 2; ELSE A = 1;
    DO CTR = A TO VARC(0);
        IF NOT DIGIT(VARC(CTR)) THEN
            DO;
                CALL PRINT$ERROR('NN');
                RETURN A$CTR;
            END;
        ELSE A$CTR = SHL(ACTR,3) + SHL(ACTR,1) +
            VARC(CTR) - '0';
    END;
    RETURN ACTR;
END CONVERT$INTEGER;

BACKSTUFF: PROC (ADD1,ADD2);
    DCL (ADD1,ADD2) ADDRESS;
    CALL BYTE$OUT(BST);
    CALL ADDR$OUT(ADD1);
    CALL ADDR$OUT(ADD2);
END BACK$STUFF;

CHK$NXT$SENTENCE: PROC;
    IF NEXT$ADDRESS <> 0 THEN
        DO;
            CALL BACKSTUFF(NEXT$ADDRESS,NEXT$AVAILAPLE);
            NEXT$ADDRESS = 0;
```

```
        END;
END CHK$NXT$SENTENCE;

UNRES$BRANCH: PROC;
    CALL SET$VALUE(NEXT$AVAILABLE + 1);
    CALL ONE$ADDR$OPP(BRN,0);
    CALL SET$VALUE2(NEXT$AVAILABLE);
END UNRES$BRANCH;

BACK$COND: PROC;
    CALL BACKSTUFF(VALUE(SP - 1),NEXT$AVAILABLE);
END BACK$COND;

SET$BRANCH: PROC;
    CALL SET$VALUE(NEXT$AVAILABLE);
    CALL CODE$ADDRESS(0);
END SET$BRANCH;

KEEP$VALUES:  PROC;
    CALL SET$VALUE(VALUE(SP));
    CALL SET$VALUE2(VALUE2(SP));
END KEEP$VALUES;

CARRAGE$CONTROL: PROC;
    WRITE$BEFORE,WRITE$AFTER = FALSE;
    CALL CODE$BYTE(PAG);
    CALL CODE$ADDRESS(GET$FCB$ADDR);
    CALL CODE$BYTE(VALUE(SP));
END CARRAGE$CONTROL;

STD$ATTRIBUTES: PROC(TYPE);
     DCL TYPE BYTE;
     CALL CODE$ADDRESS(GET$FCB$ADDR);
     CUR$SYM = GET$ADDRESS;
     CALL CODE$ADDRESS(GET$ADDRESS);
     CALL CODE$ADDRESS(GET$LENGTH);
     IF TYPE = 0 THEN RETURN;
     CUR$SYM = GET$FCB$ADDR;
     CUR$SYM = SYMBOL$ADDR(REL$ID);
     CALL CODE$ADDRESS(GET$ADDRESS);
     CALL CODE$BYTE(GET$LENGTH);
END STD$ATTRIBUTES;

WRITE$A$RECORD: PROC;
    DCL TEMP$SYM ADDRESS;
    IF GET$LEVEL <> 1 THEN CALL PRINT$ERROR('WL');
    ELSE
        DO;
            TEMP$SYM = CUR$SYM;
            CUR$SYM = GET$FCB$ADDR;
            IF (CTR := GET$TYPE) <> 1 AND
```

```
            (WRITE$BEFORE OR WRITE$AFTER) THEN
            CALL PRINT$ERROR('CC');
        IF CTR = 1 THEN
            DO;
                IF WRITE$AFTER THEN CALL CARRAGE$CONTROL;
                CALL CODE$BYTE(WTF);
                CALL STD$ATTRIBUTES(0);
                IF WRITE$BEFORE THEN
                    DO;
                        CUR$SYM = GET$FCB$ADDR;
                        CALL CARRAGE$CONTROL;
                    END;
            END;
        ELSE IF CTR = 2 THEN
            DO;
                CALL CODE$BYTE(WRS);
                CALL STD$ATTRIBUTES(1);
            END;
        ELSE IF CTR = 3 THEN
            DO;
                CALL CODE$BYTE(WRR);
                CALL STD$ATTRIBUTES(1);
            END;
        ELSE IF CTR = 4 THEN
            DO;
                CALL CODE$BYTE(WVL);
                CALL CODE$ADDRESS(GET$FCB$ADDR);
                CUR$SYM = TEMP$SYM;
                CALL CODE$ADDRESS(GET$ADDRESS);
                CALL CODE$ADDRESS(GET$LENGTH);
            END;
        ELSE CALL PRINT$ERROR('FT');
    END;
END WRITE$A$RECORD;

READ$A$FILE: PROC;
    IF (CTR := GET$TYPE) = 1 THEN
        DO;
            CALL CODE$BYTE(RDF);
            CALL STD$ATTRIBUTES(0);
        END;
    ELSE IF CTR = 2 THEN
        DO;
            CALL CODE$BYTE(RRS);
            CALL STD$ATTRIBUTES(1);
        END;
    ELSE IF CTR = 3 THEN
        DO;
            CALL CODE$BYTE(RRR);
            CALL STD$ATTRIBUTES(1);
        END;
```

```
    ELSE IF CTR = 4 THEN
        DO;
            CALL CODE$BYTE(RVL);
            CALL CODE$ADDRESS(GET$FCB$ADDR);
            CALL CODE$ADDRESS(GET$LENGTH);
            CUR$SYM = GET$ADDRESS;
            CALL CODE$ADDRESS(GET$ADDRESS);
        END;
    ELSE CALL PRINT$ERROR('FT');
END READ$A$FILE;

ARITHMETIC$TYPE: PROC BYTE;
    IF ((L$TYPE := AND$OUT$OCCURS(L$TYPE)) >=
        NUMERIC$LITERAL) AND (L$TYPE <= COMP) THEN
           RETURN L$TYPE - NUMERIC$LITERAL;
    IF L$TYPE = LIT$ZERO OR L$TYPE = ALPHA$NUM THEN
        RETURN 0;
    CALL INVALID$TYPE;
    RETURN 0;
END ARITHMETIC$TYPE;

DELETE$A$FILE: PROC;
    IF (CTR := GET$TYPE) = 3 THEN
        DO;
            CALL CODE$BYTE(DLR);
            CALL STD$ATTRIBUTES(1);
        END;
    ELSE IF CTR = 2 THEN
        DO;
            CALL CODE$BYTE(DLS);
            CALL STD$ATTRIBUTES(0);
        END;
    ELSE CALL PRINT$ERROR('IT');
END DELETE$A$FILE;

REWRITE$A$RECORD: PROC;
    IF GET$LEVEL <> 1 THEN CALL PRINT$ERROR('WL');
    ELSE
        DO;
            CUR$SYM = GET$FCB$ADDR;
            IF (CTR := GET$TYPE) = 3 THEN
                DO;
                    CALL CODE$BYTE(RWR);
                    CALL STD$ATTRIBUTES(1);
                END;
            ELSE IF CTR = 2 THEN
                DO;
                    CALL CODE$BYTE(RWS);
                    CALL STD$ATTRIBUTES(0);
                END;
            ELSE CALL PRINT$ERROR('IT');
```

```
        END;
END REWRITE$A$RECORD;

ATTRIBUTES: PROC;
    CALL CODE$ADDRESS(L$ADDR);
    CALL CODE$BYTE(L$LENGTH);
    CALL CODE$BYTE(L$DEC);
END ATTRIBUTES;

LOAD$L$ID: PROC(S$PTR);
    DCL S$PTR BYTE;
    IF ((A$CTR := VALUE(S$PTR)) <= NON$NUMERIC$LIT) OR
        (A$CTR = NUMERIC$LITERAL) THEN
        DO;
            L$ADDR = VALUE2(SPTR);
            L$LENGTH = CON$LENGTH;
            L$TYPE = A$CTR;
            IF A$CTR = NUMERIC$LITERAL THEN
                L$DEC = L$DEC$TEMP;
            ELSE L$DEC = 0;
            RETURN;
        END;
    IF A$CTR <= LIT$ZERO THEN
        DO;
            L$TYPE,L$ADDR = A$CTR;
            L$DEC = 0;
            L$LENGTH = 1;
            RETURN;
        END;
    CUR$SYM = VALUE(S$PTR);
    L$TYPE = GET$TYPE;
    L$LENGTH = GET$LENGTH;
    L$DEC = GET$DECIMAL;
    IF(L$ADDR := VALUE2(S$PTR)) = 0 THEN
        L$ADDR = GET$ADDRESS;
END LOAD$L$ID;

LOAD$REG: PROC(REG$NO,PTR);
    DCL (REG$NO,PTR) BYTE;
    CALL LOAD$L$ID(PTR);
    CALL CODE$BYTE(LOD+ARITHMETIC$TYPE);
    CALL ATTRIBUTES;
    CALL CODE$BYTE(REG$NO);
END LOAD$REG;

STORE$REG: PROC(PTR);
    DCL PTR BYTE;
    CALL LOAD$L$ID(PTR);
    CALL CODE$BYTE(STO + ARITHMETIC$TYPE - 1);
    CALL ATTRIBUTES;
END STORE$REG;
```

```
STORE$CONSTANT: PROC ADDRESS;
    IF(MAX$INT$MEM := MAX$INT$MEM - VARC(Ø)) < NEXT$AVAILABLE
        THEN CALL FATAL$ERROR('MO');
    CALL BYTE$OUT(INT);
    CALL ADDR$OUT(MAX$INT$MEM);
    CALL ADDR$OUT(CON$LENGTH := VARC(Ø));
    DO CTR = 1 TO CON$LENGTH;
        CALL BYTE$OUT(VARC(CTR));
    END;
    RETURN MAX$INT$MEM;
END STORE$CONSTANT;

NUMERIC$LIT: PROC BYTE;
    DCL CHAR BYTE;
    L$DEC$TEMP = Ø;
    DO CTR = 1 TO VARC(Ø);
        IF NOT( DIGIT(CHAR := VARC(CTR))
            OR (CHAR = '-') OR (CHAR = '+')
            OR (CHAR = '.')) THEN RETURN FALSE;
        IF CHAR = '.' THEN
            L$DEC$TEMP=VARC(Ø)-CTR;
    END;
    RETURN TRUE;
END NUMERIC$LIT;

ALPHA$LIT: PROC BYTE;
    DO CTR = 1 TO VARC(Ø);
        IF NOT(LETTER(VARC(CTR))) THEN RETURN FALSE;
    END;
    RETURN TRUE;
END ALPHA$LIT;

ROUND$STORE: PROC;
    IF VALUE(SP) <> Ø THEN
        DO;
            CALL CODE$BYTE(RND);
            CALL CODE$BYTE(L$DEC);
        END;
    CALL STORE$REG(SP - 1);
END ROUND$STORE;

ADD$SUB: PROC(INDEX);
    DCL INDEX BYTE;
    CALL LOAD$REG(1,SP - 1);
    CALL CODE$BYTE(ADD + INDEX);
    CALL ROUND$STORE;
END ADD$SUB;

MULT$DIV: PROC(INDEX);
    DCL INDEX BYTE;
```

```
    CALL LOAD$REG(0,MPP1);
    CALL LOAD$REG(1,SP - 1);
    CALL CODE$BYTE(MUL + INDEX);
    CALL ROUND$STORE;
END MULT$DIV;

CHECK$SUBSCRIPT: PROC;
    DCL (TEMP,TEMP$ADDR) ADDRESS;
    CUR$SYM = VALUE(MP);
    IF GET$TYPE < MULT$OCCURS THEN
        DO;
            CALL PRINT$ERROR('IS');
            RETURN;
        END;
    IF NUMERIC$LIT THEN
      DO;
        TEMP$ADDR = GET$ADDRESS;
        IF (TEMP := GET$PREV$OCCURS) <> 0 THEN
            CUR$SYM = TEMP;
        CALL SET$VALUE2
          (TEMP$ADDR + (GET$LENGTH * (CONVERT$INTEGER - 1)));
        RETURN;
      END;
    CALL ONE$ADDR$OPP(SCR,GET$ADDRESS);
    IF (TEMP := GET$PREV$OCCURS) <> 0 THEN
        CUR$SYM = TEMP;
    CALL CODE$ADDRESS(GET$LENGTH);
    CUR$SYM = MATCH;
    IF ((CTR := GET$TYPE) < NUMERIC) OR (CTR > COMP) THEN
        CALL PRINT$ERROR('TE');
    CALL CODE$ADDRESS(GET$ADDRESS);
    CALL CODE$BYTE(GET$LENGTH);
    CALL CODE$BYTE(SUB$CNT);
    CALL SET$VALUE2(SUB$IND);
END CHECK$SUBSCRIPT;

LOAD$LABEL: PROC;
    CUR$SYM = VALUE(MP);
    IF (A$CTR := GET$ADDRESS) <> 0 THEN
        CALL BACK$STUFF(A$CTR,VALUE2(MP));
    CALL SET$ADDRESS(VALUE2(MP));
    IF GET$TYPE <> UNRESOLVED THEN
        CALL PRINT$ERROR('DD');
    CALL SET$TYPE(LABEL$TYPE);
    IF (A$CTR := GET$FCB$ADDR) <> 0 THEN
        CALL BACK$STUFF(A$CTR,NEXT$AVAILABLE);
    SYMBOL$ADDR(FCB$ADDR) = NEXT$AVAILABLE;
    CALL ONE$ADDR$OPP(RET,0);
END LOAD$LABEL;

LOAD$SEC$LABEL: PROC;
```

```
    A$CTR = VALUE(MP);
    CALL SET$VALUE(HOLD$SECTION);
    HOLD$SECTION = A$CTR;
    A$CTR = VALUE2(MP);
    CALL SET$VALUE2(HOLD$SEC$ADDR);
    HOLD$SEC$ADDR = A$CTR;
    CALL LOAD$LABEL;
END LOAD$SEC$LABEL;

LABEL$ADDR$OFFSET: PROC (ADDR, HOLD, OFFSET) ADDRESS;
    DCL ADDR ADDRESS;
    DCL (HOLD, OFFSET, CTR) BYTE;
    CUR$SYM = ADDR;
    IF(CTR := GET$TYPE) = LABEL$TYPE THEN
        DO;
            IF HOLD THEN RETURN GET$ADDRESS;
            RETURN GET$FCB$ADDR;
        END;
    IF CTR <> UNRESOLVED THEN CALL INVALID$TYPE;
    IF HOLD THEN
        DO;
            A$CTR = GET$ADDRESS;
            CALL SET$ADDRESS(NEXT$AVAILABLE + OFFSET);
            RETURN A$CTR;
        END;
    A$CTR = GET$FCB$ADDR;
    SYMBOL$ADDR(FCB$ADDR) = NEXT$AVAILABLE + OFFSET;
    RETURN A$CTR;
END LABEL$ADDR$OFFSET;

LABEL$ADDR: PROC (ADDR, HOLD) ADDRESS;
    DCL ADDR ADDRESS,
           HOLD BYTE;
    RETURN LABEL$ADDR$OFFSET (ADDR, HOLD, 1);
END LABEL$ADDR;

CODE$FOR$DISPLAY: PROC (POINT);
    DCL POINT BYTE;
    CALL LOAD$L$ID(POINT);
    CALL ONE$ADDR$OPP(DIS,L$ADDR);
    CALL CODE$BYTE(L$LENGTH);
    IF DISPLAY$FLAG THEN CALL CODE$BYTE(1);
    ELSE CALL CODE$BYTE(Ø);
    DISPLAY$FLAG = FALSE;
END CODE$FOR$DISPLAY;

A$AN$TYPE: PROC BYTE;
    RETURN (L$TYPE >= ALPHA) AND (L$TYPE <= LIT$QUOTE);
END A$AN$TYPE;

NOT$INTEGER: PROC BYTE;
```

```
    RETURN L$DEC <> 0;
END NOT$INTEGER;

NUMERIC$TYPE: PROC BYTE;
    RETURN ((L$TYPE >= NUMERIC$LITERAL) AND (L$TYPE <= COMP))
           OR (L$TYPE=LIT$ZERO);
END NUMERIC$TYPE;

GEN$COMPARE: PROC;
    DCL (H$TYPE,H$DEC) BYTE,(H$ADDR,H$LENGTH) ADDRESS;
    CALL LOAD$L$ID(MP);
    L$TYPE = AND$OUT$OCCURS(L$TYPE);
    IF COND$TYPE = 3 THEN  /* COMPARE FOR NUMERIC */
        DO;
            IF L$TYPE = ALPHA OR (L$TYPE > COMP) THEN
                CALL INVALID$TYPE;
            CALL SET$VALUE2(NEXT$AVAILABLE);
            IF L$TYPE = NUMERIC THEN CALL CODE$BYTE(CNU);
            ELSE CALL CODE$BYTE(CNS);
            CALL CODE$ADDRESS(L$ADDR);
            CALL CODE$ADDRESS(L$LENGTH);
            CALL SET$BRANCH;
        END;
    ELSE IF COND$TYPE = 4 THEN
        DO;
            IF NUMERIC$TYPE THEN CALL INVALID$TYPE;
            CALL SET$VALUE2(NEXT$AVAILABLE);
            CALL CODE$BYTE(CAL);
            CALL CODE$ADDRESS(L$ADDR);
            CALL CODE$ADDRESS(L$LENGTH);
            CALL SET$BRANCH;
        END;
    ELSE DO;
         IF NUMERIC$TYPE THEN CTR=1;
         ELSE CTR = 0;
         H$TYPE = L$TYPE;
         H$DEC = L$DEC;
         H$ADDR = L$ADDR;
         H$LENGTH = L$LENGTH;
         CALL LOAD$L$ID(SP);
         IF NUMERIC$TYPE THEN CTR = CTR + 1;
         IF CTR = 2 THEN    /* NUMERIC COMPARE */
             DO;
                 CALL LOAD$REG(0,MP);
                 CALL SET$VALUE2(NEXT$AVAILABLE - 6);
                 CALL LOAD$REG(1,SP);
                 CALL CODE$BYTE(SUB);
                 CALL CODE$BYTE(RGT + COND$TYPE);
                 CALL SET$BRANCH;
             END;
         ELSE DO;
```

```
            /* ALPHA NUMERIC COMPARE */
            IF (H$TYPE = COMP) OR (L$TYPE = COMP) THEN
                CALL INVALID$TYPE;
            ELSE IF (H$LENGTH <> L$LENGTH) THEN
              IF NOT ((L$TYPE >= LIT$SPACE) AND
                      (L$TYPE <= LIT$ZERO)) XOR
                     ((H$TYPE >= LIT$SPACE) AND
                      (H$TYPE <= LIT$ZERO)) THEN
                 CALL INVALID$TYPE;
            ELSE IF (L$DEC <> 0) OR (H$DEC <> 0) THEN
                IF NOT ((L$TYPE = NUM$ED) XOR
                        (H$TYPE = NUM$ED)) THEN
                   CALL INVALID$TYPE;
            CALL SET$VALUE2(NEXT$AVAILABLE);
            CALL CODE$BYTE(SGT+COND$TYPE);
            CALL CODE$ADDRESS(H$ADDR);
            CALL CODE$ADDRESS(L$ADDR);
            CALL CODE$ADDRESS(H$LENGTH);
            CALL SET$BRANCH;
        END;
    END;
END GEN$COMPARE;

MOVE$TYPE: PROC BYTE;
    DCL
    HOLD$TYPE BYTE,
    ALPHA$NUM$MOVE      LIT  '0',
    A$N$ED$MOVE         LIT  '1',
    NUMERIC$MOVE        LIT  '2',
    N$ED$MOVE           LIT  '3';
    L$TYPE = AND$OUT$OCCURS(L$TYPE);
    IF((HOLD$TYPE := AND$OUT$OCCURS(GET$TYPE)) = GROUP) OR
        (L$TYPE = GROUP)
         THEN RETURN ALPHA$NUM$MOVE;
    IF HOLD$TYPE = ALPHA THEN
         IF A$AN$TYPE OR (L$TYPE = A$ED) OR (L$TYPE = A$N$ED)
              OR ((ALPHA$LIT$FLAG) AND
              (L$TYPE = NON$NUMERIC$LIT))
              THEN RETURN ALPHA$NUM$MOVE;
    IF HOLD$TYPE=ALPHA$NUM THEN
        DO;
             IF NOT$INTEGER AND (L$TYPE <> NUM$ED) THEN
                 CALL INVALID$TYPE;
             RETURN ALPHA$NUM$MOVE;
        END;
    IF (HOLD$TYPE >= NUMERIC) AND (HOLD$TYPE <= COMP) THEN
        DO;
             IF (L$TYPE = ALPHA) OR (L$TYPE > COMP) THEN
                 CALL INVALID$TYPE;
             RETURN NUMERIC$MOVE;
        END;
```

```
    IF HOLD$TYPE = A$N$ED THEN
        DO;
             IF NOT$INTEGER AND (L$TYPE <> NUM$ED) THEN
                 CALL INVALID$TYPE;
             RETURN A$N$ED$MOVE;
        END;
    IF HOLD$TYPE = A$ED THEN
         IF A$AN$TYPE OR (L$TYPE > COMP) OR
            (L$TYPE = NON$NUMERIC$LIT)
             THEN RETURN A$N$ED$MOVE;
    IF HOLD$TYPE = NUM$ED THEN
         IF NUMERIC$TYPE OR (L$TYPE = ALPHA$NUM) THEN
              RETURN N$ED$MOVE;
    CALL INVALID$TYPE;
    RETURN 0;
END MOVE$TYPE;

GEN$MOVE:PROC;
    DCL (ADDR1,EXTRA,LENGTH1) ADDRESS;

    ADD$ADD$LEN: PROC;
        CALL CODE$ADDRESS(ADDR1);
        CALL CODE$ADDRESS(L$ADDR);
        CALL CODE$ADDRESS(L$LENGTH);
    END ADD$ADD$LEN;

    CODE$FOR$EDIT: PROC;
        CALL ADD$ADD$LEN;
        CALL CODE$ADDRESS(GET$FCB$ADDR);
        CALL CODE$ADDRESS(LENGTH1);
    END CODE$FOR$EDIT;

    CALL LOAD$L$ID(MPP1);
    CUR$SYM=VALUE(SP);
    IF (ADDR1 := VALUE2(SP)) = 0 THEN ADDR1 = GET$ADDRESS;
    LENGTH1 = GET$LENGTH;
    DO CASE MOVE$TYPE;
         /* ALPHA NUMERIC MOVE */
         DO;
              IF LENGTH1 > L$LENGTH THEN
                  EXTRA = LENGTH1 - L$LENGTH;
              ELSE DO;
                        EXTRA = 0;
                        L$LENGTH = LENGTH1;
                   END;
              CALL CODE$BYTE(MOV);
              CALL ADD$ADD$LEN;
              CALL CODE$ADDRESS(EXTRA);
         END;
         /* ALPHA NUMERIC EDITED */
         DO;
```

```
            CALL CODE$BYTE(MED);
            CALL CODE$FOR$EDIT;
        END;
        /* NUMERIC MOVE */
        DO;
            CALL LOAD$REG(2,MPP1);
            CALL STORE$REG(SP);
        END;
        /* NUMERIC EDITED MOVE */
        DO;
            CALL CODE$BYTE(MNE);
            CALL CODE$FOR$EDIT;
            CALL CODE$BYTE(L$DEC);
            CALL CODE$BYTE(GET$DECIMAL);
        END;
    END;
END GEN$MOVE;

CODE$GEN: PROC(PRODUCTION);
    DCL PRODUCTION BYTE;
    IF PRINT$PROD THEN
        DO;
            CALL CRLF;
            CALL PRINTCHAR(POUND);
            CALL PRINT$NUMBER(PRODUCTION);
        END;
    DO CASE PRODUCTION;
  /*  P R O D U C T I O N S */
    ;    /* CASE Ø NOT USED  */
/*       1   <P-DIV> ::= PROCEDURE DIVISION <USING> .       */
/*       1               <PROC-BODY>                        */
    DO;
        COMPILING = FALSE;
        IF SECTION$FLAG THEN CALL LOAD$SEC$LABEL;
    END;
/*       2   <USING> ::= USING <ID-STRING>                  */
    IF VALUE(MP - 1) = Ø THEN
        DO I = Ø TO ID$PTR;
            CURSYM = ID$STACK(I);
            CALL SET$ADDRESS(13 + I);
        END;
    ELSE
        DO;
            CALL CODE$BYTE(PAR);
            CALL CODE$ADDRESS(ID$PTR + 1);
            DO I = Ø TO ID$PTR;
                CUR$SYM = ID$STACK(I);
                CALL CODE$ADDRESS(GET$ADDRESS);
            END;
        END;
/*       3             \! <EMPTY>                          */
```

```
    ;     /* NO ACTION REQUIRED */
/*        4   <ID-STRING> ::= <ID>                                    */
    ID$STACK(ID$PTR := 0) = VALUE(SP);
/*        5                \! <ID-STRING> <ID>                        */
    DO;
        IF(ID$PTR := IDPTR + 1) = 20 THEN
            DO;
                CALL PRINT$ERROR('ID');
                ID$PTR=19;
            END;
        ID$STACK(ID$PTR)=VALUE(SP);
    END;
/*        6   <PROC-BODY> ::= <PARAGRAPH>                             */
    ;     /* NO ACTION REQUIRED */
/*        7                   \! <PROC-BODY> <PARAGRAPH>              */
    ;     /* NO ACTION REQUIRED */
/*        8   <PARAGRAPH> ::= <ID> .                                  */
    ;     /* NO ACTION REQUIRED */
/*        9                   \! <ID> . <SENTENCE-LIST>               */
    DO;
      . IF SECTION$FLAG = 0 THEN SECTION$FLAG = 2;
        CALL LOAD$LABEL;
    END;
/*       10                   \! <ID> SECTION .                       */
    DO;
        IF SECTION$FLAG<>1 THEN
            DO;
                IF SECTION$FLAG = 2 THEN
                    CALL PRINT$ERROR('PF');
                SECTION$FLAG = 1;
                HOLD$SECTION = VALUE(MP);
                HOLD$SEC$ADDR = VALUE2(MP);
            END;
        ELSE CALL LOAD$SEC$LABEL;
    END;
/*       11   <SENTENCE-LIST> ::= <SENTENCE> .                       */
    CALL CHK$NXT$SENTENCE;
/*       12                       \! <SENTENCE-LIST>                 */
/*       12                          <SENTENCE> .                    */
    CALL CHK$NXT$SENTENCE;
/*       13   <SENTENCE> ::= <IMPERATIVE>                            */
    ;     /* NO ACTION REQUIRED */
/*       14                  \! <CONDITIONAL>                        */
    ;     /* NO ACTION REQUIRED */
/*       15                  \! ENTER <ID> <OPT-ID>                  */
    CALL PRINT$ERROR('NI');
/*       16   <IMPERATIVE> ::= ACCEPT <SUBID>                        */
    DO;
        CALL LOAD$L$ID(SP);
        CALL ONE$ADDR$OPP(ACC,L$ADDR);
        CALL CODE$BYTE(L$LENGTH);
```

```
    END;
/*      17                  \! <ARITHMETIC>                 */
    ;    /* NO ACTION REQUIRED */
/*      18                  \! CALL <CALL-LIT> <USING>      */
    DO;
        CURSYM = VALUE(MPP1);
        CALL CODE$BYTE(SBR);
        DO I = 1 TO 8;
            IF I <= GET$P$LENGTH THEN
                CALL BYTE$OUT(SYMBOL(START$NAME + I));
            ELSE CALL BYTE$OUT(20H);
        END;
        CALL INC$COUNT(6);
    END;
/*      19                  \! CLOSE <CLOSE-LST>            */
    DO;
        DCL TYPE BYTE;
        IF ((TYPE := GET$TYPE) > 0) AND (TYPE < 5) THEN
            CALL ONE$ADDR$OPP(CLS,GET$FCB$ADDR);
        ELSE CALL PRINT$ERROR('CE');
    END;
/*      19                  \! <FILE-ACT>                   */
    ;    /* NO ACTION REQUIRED */
/*      21                  \! DISPLAY <DISPLAY-LST>        */
    ;    /* NO ACTION REQUIRED */
/*      22                  \! DISPLAY <DISPLAY-LST> WITH   */
/*      22                     NO ADVANCING                 */
    ;    /* NO ACTION REQUIRED-NOT IMPLEMENTED */
/*      23                  \! EXIT <PROGRAM-ID>            */
    CALL CODE$BYTE(EXT);
/*      24                  \! GO <ID>                      */
    CALL ONE$ADDR$OPP(BRN,LABEL$ADDR(VALUE(SP),1));
/*      25                  \! GO <ID-STRING> DEPENDING     */
/*      25                     <ID>                         */
    DO;
        CALL CODE$BYTE(GDP);
        CALL CODE$BYTE(ID$PTR + 1);
        CUR$SYM = VALUE(SP);
        CALL CHK$UD$VAR(SP);
        CALL CODE$BYTE(GET$LENGTH);
        CALL CODE$ADDRESS(GET$ADDRESS);
        DO CTR = 0 TO ID$PTR;
           CALL CODE$ADDRESS
              (LABEL$ADDR$OFFSET(ID$STACK(CTR),1,0));
        END;
    END;
/*      26                  \! MOVE <LIT/ID> TO <SUBID>     */
    CALL GEN$MOVE;
/*      27                  \! OPEN <ACT-LST>               */
    ;    /* NO ACTION REQUIRED */
/*      28                  \! PERFORM <ID> <THRU> <FINISH>*/
```

```
    DO;
        DCL (ADDR2,ADDR3) ADDRESS;
        IF VALUE(SP - 1) = 0 THEN
            ADDR2 = LABEL$ADDR$OFFSET(VALUE(MPP1),0,3);
        ELSE ADDR2 = LABEL$ADDR$OFFSET(VALUE(SP-1),0,3);
        IF (ADDR3 := VALUE2(SP)) = 0 THEN
            ADDR3 = NEXT$AVAILABLE + 7;
        ELSE CALL BACKSTUFF(VALUE(SP),NEXT$AVAILABLE + 7);
        CALL ONE$ADDR$OPP(PER,LABEL$ADDR(VALUE(MPP1),1));
        CALL CODE$ADDRESS(ADDR2);
        CALL CODE$ADDRESS(ADDR3);
    END;
/*      29                  \! STOP <TERMINATE>                 */
    DO;
        IF VALUE(SP) = 0 THEN CALL CODE$BYTE(STP);
        ELSE IF (VALUE(SP) < LIT$SPACE) OR
                (VALUE(SP) > LIT$ZERO) THEN
            DO;
                CALL ONE$ADDR$OPP(STD,VALUE2(SP));
                CALL CODE$BYTE(CON$LENGTH);
            END;
        ELSE
            DO;
                CALL ONE$ADDR$OPP(STD,VALUE(SP));
                CALL CODE$BYTE(1);
            END;
    END;
/*      30   <CLOSE-LST> ::= <ID>                               */
    ;    /* NO ACTION REQUIRED */
/*      31                  \! <CLOSE-LST> <ID>                 */
    ;    /* NO ACTION REQUIRED-NOT IMPLEMENTED */
/*      32   <DISPLAY-LST> ::= <LIT/ID>                         */
    CALL CODE$FOR$DISPLAY(SP);
/*      33                    \! <DISPLAY-LST> <LIT/ID>         */
    DO;
        DISPLAY$FLAG = TRUE;
        CALL CODE$FOR$DISPLAY(SP);
    END;
/*      34   <ACT-LST> ::= <TYPE-ACTION> <OPEN-LST>             */
    DO;
        DCL TYPE BYTE;
        TYPE = GET$TYPE;
        IF (TYPE = 1 OR TYPE = 4) AND (VALUE(MP) <> 2) THEN
            CALL ONE$ADDR$OPP(OPN + VALUE(MP),GET$FCB$ADDR);
        ELSE
          IF (TYPE = 2 OR TYPE = 3) THEN
            CALL ONE$ADDR$OPP(OPN + VALUE(MP),GET$FCB$ADDR);
          ELSE CALL PRINT$ERROR('0E');
    END;
/*      35                \! <ACT-LST> <TYPE-ACTION>            */
/*      35                   <OPEN-LST>                         */
```

```
    ;     /* NO ACTION REQUIRED-NOT IMPLEMENTED */
/*       36   <OPEN-LST> ::= <ID>                                    */
    ;     /* NO ACTION REQUIRED */
/*       37                  \! <OPEN-LST> <ID>                      */
    ;     /* NO ACTION REQUIRED-NOT IMPLEMENTED */
/*       38   <FINISH> ::= <L/ID> TIMES                              */
   DO;
       CALL LOAD$L$ID(MP);
       CALL ONE$ADDR$OPP(LDI,L$ADDR);
       CALL CODE$BYTE(L$LENGTH);
       CALL SET$VALUE2(NEXT$AVAILABLE);
       CALL ONE$ADDR$OPP(DEC,0);
       CALL SET$VALUE(NEXT$AVAILABLE);
       CALL CODE$ADDRESS(0);
   END;
/*       39                      \! <STOPCONDITION>                  */
    CALL KEEP$VALUES;
/*       40                      \! <VARYING> <ITERATION>            */
/*       40                         <STOPCONDITION>                  */
    CALL KEEP$VALUES;
/*       41                      \! <EMPTY>                          */
    ;     /* NO ACTION REQUIRED */
/*       42   <STOPCONDITION> ::= UNTIL <CONDITION>                  */
    CALL KEEP$VALUES;
/*       43   <VARYING> ::= VARYING <SUBID>                          */
    CALL KEEP$VALUES;
/*       44   <ITERATION> ::= <FROM> <BY>                            */
    ;     /* NO ACTION REQUIRED */
/*       45   <FROM> ::= FROM <L/ID>                                 */
    DO;
        CALL LOAD$REG(2,SP);
        CALL STORE$REG(MP - 1);
    END;
/*       46   <BY> ::= BY <L/ID>                                     */
    DO;
        CALL LOAD$REG(0,MP - 2);
        CALL LOAD$REG(1,SP);
        CALL CODE$BYTE(ADD);
        CALL STORE$REG(MP - 2);
    END;
/*        47   <CONDITIONAL> ::= <ARITHMETIC> <SIZE-ERROR>           */
/*        47                     <IMPERATIVE>                        */
    CALL BACK$COND;
/*       48                      \! <FILE-ACT> <INVALID>             */
/*       48                         <IMPERATIVE>                     */
    CALL BACK$COND;
/*       49                      \! <READ-ID> <SPECIAL>              */
/*       49                         <IMPERATIVE>                     */
    CALL BACK$COND;
/*       50                      \! <IF-NONTERMINAL>                 */
/*       50                         <CONDITION> <IF-LST> <ELSE>      */
```

```
/*      50                          <IF-LST> END-IF                 */
    DO;
        CALL BACKSTUFF(VALUE(MPP1),VALUE2(SP - 3));
        CALL BACKSTUFF(VALUE(SP - 3),NEXT$AVAILABLE);
    END;
/*      51                        \! <IF-NONTERMINAL>               */
/*      51                           <CONDITION>                    */
/*      51                           <IF-LST> END-IF                */
    CALL BACKSTUFF(VALUE(MPP1),NEXT$AVAILABLE);
/*      52   <IF-LST> ::= <STMT-LST>                                */
    ;    /* NO ACTION REQUIRED */
/*      53                \! NEXT SENTENCE                          */
    DO;
        CALL ONE$ADDR$OPP(BRN,NEXT$ADDRESS);
        NEXT$ADDRESS = NEXT$AVAILABLE - 2;
    END;
/*      54   <ELSE> ::= ELSE                                        */
    DO;
         VALUE(SP - 1) = NEXT$AVAILABLE + 1;
         CALL ONE$ADDR$OPP(BRN,0);
         VALUE2(SP - 1) = NEXT$AVAILABLE;
    END;
/*      55   <ARITHMETIC> ::= ADD <ADD-LST> TO <SUBID>              */
/*      55                       <ROUND>                            */
    CALL ADD$SUB(0);
/*      56                    \! ADD <ADD-LST> GIVING <SUBID>       */
/*      56                       <ROUND>                            */
    DO;
        IF VALUE(MP) = 0 THEN CALL PRINT$ERROR('IG');
        CALL ROUND$STORE;
    END;
/*      57                    \! DIVIDE <L/ID> INTO <SUBID>         */
/*      57                       <ROUND>                            */
    CALL MULT$DIV(1);
/*      58                    \! DIVIDE <L/ID> BY <SUBID>           */
/*      58                       GIVING <SUBID> <ROUND>             */
    CALL PRINT$ERROR('NI');
/*      59                    \! DIVIDE <L/ID> INTO <SUBID>         */
/*                               GIVING <SUBID> <ROUND>             */
    CALL PRINT$ERROR('NI');
/*      60                    \! MULTIPLY <L/ID> BY <SUBID>         */
/*      60                       <ROUND>                            */
    CALL MULT$DIV(0);
/*      61                    \! MULTIPLY <L/ID> BY <SUBID>         */
/*      61                       GIVING <SUBID> <ROUND>             */
    CALL PRINT$ERROR('NI');
/*      62                   \! SUBTRACT <SUB-LST> FROM             */
/*      62                      <SUBID> <ROUND>                     */
    CALL ADD$SUB(1);
/*      63                   \! SUBTRACT <SUB-LST> GIVING           */
/*      63                      <SUBID> <ROUND>                     */
```

```
    DO;
        IF VALUE(MP) = 0 THEN CALL PRINT$ERROR('IG');
        CALL ROUND$STORE;
    END;
/*      64                  \! COMPUTE <SUBID> = <ARITH-EXP>*/
    CALL PRINT$ERROR('NI');
/*      65   <ADD-LST> ::= <L/ID>                          */
    CALL LOAD$REG(0,SP);
/*      66                \! <ADD-LST> <L/ID>              */
    DO;
        CALL LOAD$REG(1,SP);
        CALL CODE$BYTE(ADD);
        CALL CODE$BYTE(STI);
        VALUE(MP - 1) = 1;
    END;
/*      67   <SUB-LST> ::= <L/ID>                          */
    CALL LOAD$REG(0,SP);
/*      68                \! <SUB-LST <L/ID>               */
    DO;
        CALL LOAD$REG(1,SP);
        CALL CODE$BYTE(ADD);
        CALL CODE$BYTE(STI);
        VALUE(MP - 1) = 1;
    END;
/*      69   <ARITH-EXP> ::= <TERM>                        */
    ;    /* NO ACTION REQUIRED-NOT IMPLEMENTED */
/*      70                  \! <ARITH-EXP> + <TERM>        */
    ;    /* NO ACTION REQUIRED-NOT IMPLEMENTED */
/*      71                  \! <ARITH-EXP> - <TERM>        */
    ;    /* NO ACTION REQUIRED-NOT IMPLEMENTED */
/*      72                  \! + <TERM>                    */
    ;    /* NO ACTION REQUIRED-NOT IMPLEMENTED */
/*      73                  \! - <TERM>                    */
    ;    /* NO ACTION REQUIRED-NOT IMPLEMENTED */
/*      74   <TERM> ::= <PRIMARY>                          */
    ;    /* NO ACTION REQUIRED-NOT IMPLEMENTED */
/*      75             \! <TERM> * <PRIMARY>               */
    ;    /* NO ACTION REQUIRED-NOT IMPLEMENTED */
/*      76             \! <TERM> / <PRIMARY>               */
    ;    /* NO ACTION REQUIRED-NOT IMPLEMENTED */
/*      77   <PRIMARY> ::= <PRIM-ELEM>                     */
    ;    /* NO ACTION REQUIRED-NOT IMPLEMENTED */
/*      78                \! <PRIMARY> ** <PRIM-ELEM>      */
    ;    /* NO ACTION REQUIRED-NOT IMPLEMENTED */
/*      79   <PRIM-ELEM> ::= <L/ID>                        */
    ;    /* NO ACTION REQUIRED-NOT IMPLEMENTED */
/*      80                  \! ( <ARITH-EXP> )             */
    ;    /* NO ACTION REQUIRED-NOT IMPLEMENTED */
/*      81   <FILE-ACT> ::= DELETE <ID>                    */
    CALL DELETE$A$FILE;
/*      82                  \! REWRITE <ID>                */
```

```
    CALL REWRITE$A$RECORD;
/*      83                  \! WRITE <ID> <SPECIAL-ACT>          */
    CALL WRITE$A$RECORD;
/*      84   <CONDITION> ::= <BTERM>                             */
    ;    /* NO ACTION REQUIRED */
/*      85                   \! <CONDITION> OR <BTERM>           */
    ;    /* NO ACTION REQUIRED-NOT IMPLEMENTED */
/*      86   <BTERM> ::= <BPRIM>                                 */
    ;    /* NO ACTION REQUIRED */
/*      87                   \! <BTERM> AND <BPIRM>              */
    ;    /* NO ACTION REQUIRED-NOT IMPLEMETED */
/*      88   <BPRIM> ::= <LIT/ID>                                */
    ;    /* NO ACTION REQUIRED */
/*      89                   \! <LIT/ID> <NOT> <COND-TYPE>       */
    DO;
        IF IF$FLAG THEN
            DO;
                IF$FLAG = NOT IF$FLAG; /* RESET IF$FLAG */
                CALL CODE$BYTE(NEG);
            END;
        CALL GEN$COMPARE;
    END;
/*      90                \! ( <BTERM> )                         */
    ;    /* NO ACTION REQUIRED-NOT IMPLEMENTED */
/*      91   <COND-TYPE> ::= NUMERIC                             */
    COND$TYPE = 3;
/*      92                   \! ALPHABETIC                       */
    COND$TYPE = 4;
/*      93                   \! <COMPARE> <LIT/ID>               */
    CALL KEEP$VALUES;
/*      94   <NOT> ::= NOT                                       */
    IF NOT IF$FLAG THEN
        CALL CODE$BYTE(NEG);
    ELSE IF$FLAG = NOT IF$FLAG;              /* RESET IF$FLAG */
/*      95              \! <EMPTY>                               */
    ;    /* NO ACTION REQUIRED */
/*      96   <COMPARE> ::= GREATER                               */
    COND$TYPE = 0;
/*      97                  \! LESS                              */
    COND$TYPE = 1;
/*      98                  \! EQUAL                             */
    COND$TYPE = 2;
/*      99                  \! >                                 */
    COND$TYPE = 0;
/*     100                  \! <                                 */
    COND$TYPE = 1;
/*     101                  \! =                                 */
    COND$TYPE = 2;
/*     102   <ROUND> ::= ROUNDED                                 */
    CALL SET$VALUE(1);
/*     103                  \! <EMPTY>                           */
```

```
    ;     /* NO ACTION REQUIRED */
/*      104   <TERMINATE> ::= <LITERAL>                              */
    ;     /* NO ACTION REQUIRED */
/*      105                   \! RUN                                 */
    ;     /* NO ACTION REQUIRED - VALUE(SP) ALREADY ZERO */
/*      106   <SPECIAL> ::= <INVALID>                                */
    ;     /* NO ACTION REQUIRED */
/*      107                \! END                                    */
    DO;
        CALL SET$VALUE(2);
        CALL CODE$BYTE(EOR);
        CALL SET$BRANCH;
    END;
/*      108   <OPT-ID> ::= <SUBID>                                   */
    ;     /* VALUE AND VALUE2 ALREADY SET */
/*      109              \!<EMPTY>                                   */
    ;     /* VALUE ALREADY ZERO */
/*      110   <STMT-LST> ::= <IMPERATIVE>                            */
    ;     /* NO ACTION REQUIRED */
/*      111              \! <STMT-LST> <IMPERATIVE>                  */
    ;     /* NO ACTION REQUIRED */
/*      112              \! <CONDITIONAL>                            */
    ;     /* NO ACTION REQUIRED */
/*      113               \! <STMT-LST> <CONDITIONAL>                */
    ;     /* NO ACTION REQUIRED */
/*      114   <THRU> ::= THRU <ID>                                   */
    CALL KEEP$VALUES;
/*      115            \! <EMPTY>                                    */
    ;     /* NO ACTION REQUIRED */
/*      116  <INVALID> ::= INVALID                                   */
    DO;
        CALL SET$VALUE(1);
        CALL CODE$BYTE(INV);
        CALL SET$BRANCH;
    END;
/*      117   <SIZE-ERROR> ::= SIZE ERROR                            */
    DO;
        CALL CODE$BYTE(SER);
        CALL UNRES$BRANCH;
    END;
/*      118   <SPECIAL-ACT> ::= <WHEN> ADVANCING <HOW-MANY>          */
    CALL KEEP$VALUES;         /* CARRAGE CONTROL */
/*      119                   \! <EMPTY>                             */
    ;     /* NO ACTION REQUIRED */
/*      120   <WHEN> ::= BEFORE                                      */
    WRITE$BEFORE = TRUE;      /* CARRAGE CONTROL */
/*      121             \! AFTER                                     */
    WRITE$AFTER = TRUE;       /* CARRAGE CONTROL */
/*      122   <HOW-MANY> ::= <INTEGER>                               */
    ;     /* NO ACTION REQUIRED */
/*      123                \! PAGE                                   */
```

```
    CALL SET$VALUE(101);      /* CARRAGE CONTROL */
/*     124   <TYPE-ACTION> ::= INPUT                                  */
    ;     /* NO ACTION REQUIRED - VALUE(SP) ALREADY ZERO */
/*     125                   \! OUTPUT                                */
    CALL SET$VALUE(1);
/*     126                   \! I-O                                   */
    CALL SET$VALUE(2);
/*     127   <SUBID> ::= <SUBSCRIPT>                                  */
    ;     /* VALUE AND VALUE2 ALREADY SET */
/*     128              \! <ID>                                       */
    CALL CHK$UD$VAR(SP);
/*     129   <INTEGER> ::= <INPUT>                                    */
    CALL SET$VALUE(CONVERT$INTEGER);
/*     130   <ID> ::= <INPUT>                                         */
    DO;
        CALL SET$VALUE(MATCH);
        IF GET$TYPE = UNRESOLVED THEN
            CALL SET$VALUE2(NEXT$AVAILABLE);
    END;
/*     131   <L/ID> ::= <INPUT>                                       */
    DO;
        IF NUMERIC$LIT THEN
            DO;
                CALL SET$VALUE(NUMERIC$LITERAL);
                CALL SET$VALUE2(STORE$CONSTANT);
            END;
        ELSE
            DO;
                CALL SET$VALUE(MATCH);
                CALL CHK$UD$VAR(MP);
            END;
    END;
/*     132            \! <SUBSCRIPT>                                 */
    ;     /* NO ACTION REQUIRED */
/*     133            \! ZERO                                        */
    CALL SET$VALUE(LIT$ZERO);
/*     134   <SUBSCRIPT> ::= <ID> ( <SUBSCRIPT-LST> )                 */
    CALL CHECK$SUBSCRIPT;
/*     135   <SUBSCRIPT-LST> ::= <INPUT>                              */
    ;     /* NO ACTION REQUIRED */
/*     136                  \! <SUBSCRIPT-LST> , <INPUT>              */
    CALL PRINT$ERROR('NI');
/*     137   <CALL-LIT> ::= <LIT>                                     */
    CALL SET$VALUE(MATCH);
/*     138   <NN-LIT> ::= <LIT>                                       */
    DO;
        ALPHA$LIT$FLAG = ALPHA$LIT;
        CALL SET$VALUE(NON$NUMERIC$LIT);
        CALL SET$VALUE2(STORE$CONSTANT);
    END;
/*     139             \! SPACE                                       */
```

```
    CALL SET$VALUE(LIT$SPACE);
/*      140                \! QUOTE                                    */
    CALL SET$VALUE(LIT$QUOTE);
/*      141    <LITERAL> ::= <NN-LIT>                                  */
    ;     /* NO ACTION REQUIRED */
/*      142                \! <INPUT>                                  */
    DO;
          IF NOT NUMERIC$LIT THEN CALL INVALID$TYPE;
          CALL SET$VALUE(NUMERIC$LITERAL);
          CALL SET$VALUE2(STORE$CONSTANT);
    END;
/*      143                \! ZERO                                     */
    CALL SET$VALUE(LIT$ZERO);
/*      144  <LIT/ID> ::= <L/ID>                                       */
    ;     /* NO ACTION REQUIRED */
/*      145                \! <NN-LIT>                                 */
    ;     /* NO ACTION REQUIRED */
/*      146    <PROGRAM-ID> ::= <ID>                                   */
    CALL CODE$BYTE(EXT);
/*      147                    \! <EMPTY>                              */
    ;     /* NO ACTION REQUIRED */
/*      148    <READ-ID> ::= READ <ID>                                 */
    CALL READ$A$FILE;
/*      149     <IF-NONTERMINAL> ::= IF                                */
    IF$FLAG = TRUE;                  /* SET IF$FLAG */
    END;    /* END OF CASE STATEMENT */
END CODE$GEN;

GETIN1: PROC ADDRESS;
    RETURN INDEX1(STATE);
END GETIN1;

GETIN2: PROC BYTE;
    RETURN INDEX2(STATE);
END GETIN2;

INCSP: PROC;
    VALUE(SP := SP + 1),VALUE2(SP) = 0; /* CLEAR THE STACK */
    IF SP >= PSTACKSIZE THEN CALL FATAL$ERROR('SO');
END INCSP;

LOOKAHEAD: PROC;
    IF NOLOOK THEN
        DO;
            CALL SCANNER;
            NOLOOK = FALSE;
            IF PRINT$TOKEN THEN
                DO;
                    CALL CRLF;
                    CALL PRINT$NUMBER(TOKEN);
                    CALL PRINT$CHAR(' ');
```

```
                    CALL PRINT$ACCUM;
                END;
        END;
END LOOKAHEAD;

NO$CONFLICT: PROC (CSTATE) BYTE;
    DCL (CSTATE,I,J,K) ADDRESS;
    J = INDEX1(CSTATE);
    K = J + INDEX2(CSTATE) - 1;
    DO I = J TO K;
        IF READ1(I) = TOKEN THEN RETURN TRUE;
    END;
RETURN FALSE;
END NO$CONFLICT;

RECOVER: PROC BYTE;
    DCL TSP BYTE, RSTATE ADDRESS;
    DO FOREVER;
        TSP = SP;
        DO WHILE TSP <> 255;
            IF NO$CONFLICT(RSTATE := STATESTACK(TSP)) THEN
                DO;  /* STATE WILL READ TOKEN */
                    IF SP <> TSP THEN SP = TSP - 1;
                    RETURN RSTATE;
                END;
            TSP = TSP - 1;
        END;
        CALL SCANNER;  /* TRY ANOTHER TOKEN */
    END;
END RECOVER;

    /*  *  *  *  *  PROGRAM EXECUTION STARTS HERE *  *  */

    /* INITIALIZATION */
TOKEN = 80; /* PRIME THE SCANNER WITH -PROCEDURE- */
CALL MOVE(PASS1$TOP - PASS1$LEN,.DEBUGGING,PASS1$LEN);
LIST$END = .LIST$BUFF + 127;
LIST$PTR = .LIST$BUFF + LIST$PTR;
OUTPUT$END = .OUTPUT$BUFF + 127;
OUTPUT$PTR = .OUTPUT$BUFF + OUTPUT$PTR;
CALL PRINT$ERROR(FALSE); /* INITIALIZE ERROR MSG OUTPUT */

    /*  *  *  *  *  *  *  PARSER *  *  *  *  *  */

DO WHILE COMPILING;
    IF STATE <= MAXRNO THEN        /* READ STATE */
        DO;
            CALL INCSP;
            STATESTACK(SP) = STATE;  /* SAVE CURRENT STATE */
            CALL LOOKAHEAD;
            I = GETIN1;
```

```
        J = I + GETIN2 - 1;
        DO I = I TO J;
            IF READ1(I) = TOKEN THEN
                DO;
                    IF (TOKEN = INPUT$STR) OR
                       (TOKEN = LITERAL) THEN
                        DO K = 0 TO ACCUM(0);
                            VARC(K) = ACCUM(K);
                        END;
                    STATE = READ2(I);
                    NOLOOK = TRUE;
                    I = J;
                END;
            ELSE IF I = J THEN
                DO;
                    CALL PRINT$ERROR('NP');
                    CALL PRINT(.(' ERROR NEAR $'));
                    CALL PRINT$ACCUM;
                    IF (STATE := RECOVER) = 0 THEN
                        COMPILING = FALSE;
                END; /* END OF IF I = J */
        END; /* END OF I = I TO J */
    END;    /* END OF READ STATE */
ELSE IF STATE > MAXPNO THEN /* APPLY PRODUCTION STATE */
    DO;
        MP = SP - GETIN2;
        MPP1 = MP + 1;
        CALL CODE$GEN(STATE - MAXPNO);
        SP = MP;
        I = GETIN1;
        J = STATESTACK(SP);
        DO WHILE (K := APPLY1(I)) <> 0 AND J <> K;
            I = I + 1;
        END;
        IF (K := APPLY2(I)) = 0 THEN COMPILING = FALSE;
        STATE = K;
    END;
ELSE IF STATE <= MAXLNO THEN /*LOOKAHEAD STATE*/
    DO;
        I = GETIN1;
        CALL LOOKAHEAD;
        DO WHILE (K := LOOK1(I)) <> 0 AND TOKEN <> K;
            I = I + 1;
        END;
        STATE = LOOK2(I);
    END;
ELSE DO;        /*PUSH STATES*/
         CALL INCSP;
         STATESTACK(SP) = GETIN2;
         STATE = GETIN1;
     END;
```

```
END;  /* OF WHILE COMPILING */

CALL CODE$BYTE(TER);
CALL ADDR$OUT(MAX$INT$MEM);
IF NOT NO$CODE THEN
    DO;
        CALL WRITE$OUTPUT(.OUTPUT$BUFF,.OUTPUT$FCB);
        CALL CLOSE(.OUTPUT$FCB);
    END;
CALL CHECK$UNRESOLVED;
CALL CRLF;
CALL DCRLF;
DO I = 0 TO 4;
    CALL PRINT$CHAR(ERROR$CTR(I));
    CALL WRITE$TO$DISK(ERROR$CTR(I));
END;
CALL PRINT(.(' PROGRAM ERROR(S)$'));
DO WHILE LIST$PTR < LIST$END;
    CALL WRITE$TO$DISK(' ');
END;
CALL WRITE$TO$DISK(' ');
CALL CLOSE(.LIST$FCB);
CALL BOOT;
END;
```

```
$ TITLE('NPS MIRCO-COBOL COMPILER INTERP') PAGEWIDTH(80)
    PAGELENGTH(60)
INTERP: DO;

          /*    COBOL COMPILER-INTERPRETER    */

          /*    NORMALLY LOCATED AT 103H      */

       /*    GLOBAL DECLARATIONS AND LITERALS    */

DECLARE DCL        LITERALLY 'DECLARE',
        LIT        LITERALLY 'LITERALLY';
DCL     CR         LIT       '13',
        FALSE      LIT       '0',
        FOREVER    LIT       'WHILE TRUE',
        LF         LIT       '10',
        PROC       LIT       'PROCEDURE',
        SER        LIT       '12',    /* CODE FOR SIZE ERROR */
        TAB        LIT       '09H',
        TRUE       LIT       '1',
        ZONE       LIT       '80H';

            /* UTILITY VARIABLES */

DCL     A$CTR          ADDRESS,
        BASE           ADDRESS,
        BOOTER         ADDRESS              INITIAL (0000H),
        B$ADDR         BASED BASE (1)       ADDRESS,
        B$BYTE         BASED BASE (1)       BYTE,
        CALL$BASE      ADDRESS,
        CALL$PTR       BASED CALL$BASE (1) ADDRESS,
        CALL$TOP       ADDRESS,
        CTR            BYTE,
        CTR1           BYTE,
        ERROR$CTR(5)   BYTE                 INITIAL ('    0'),
        HOLD           ADDRESS,
        H$ADDR         BASED HOLD (1)       ADDRESS,
        H$BYTE         BASED HOLD (1)       BYTE,
        HI$FREE$MEM    ADDRESS,
        LOW$FREE$MEM   ADDRESS,
        HI$OFFSET      ADDRESS              INITIAL (0),
        LOW$OFFSET     ADDRESS              INITIAL (0),
        INDEX          BYTE,
        RTN$BASE       ADDRESS,
        RTN$PTR        BASED RTN$BASE (1)       ADDRESS,

            /* CODE POINTERS */
```

```
        CODE$START      ADDRESS                 INITIAL(3500H),
        PROGRAM$COUNTER ADDRESS,
        C$ADDR          BASED PROGRAM$COUNTER(1) ADDRESS,
        C$BYTE          BASED PROGRAM$COUNTER(1) BYTE,
        MAX$MEMORY      ADDRESS                 INITIAL(0B100H);

    /* * * * *   GLOBAL INPUT AND OUTPUT ROUTINES * * * * * */

DCL
CURRENT$FCB         ADDRESS,
START$OFFSET        LIT         '37';

MON1: PROC (F,A) EXTERNAL;
    DCL F BYTE, A ADDRESS;
END MON1;

MON2: PROC (F,A) BYTE EXTERNAL;
    DCL F BYTE, A ADDRESS;
END MON2;

PRINT$CHAR: PROC (CHAR);
    DCL CHAR BYTE;
    CALL MON1 (2,CHAR);
END PRINT$CHAR;

CRLF: PROC;
    CALL PRINT$CHAR(CR);
    CALL PRINT$CHAR(LF);
END CRLF;

PRINT: PROC (A);
    DCL A ADDRESS;
    CALL CRLF;
    CALL MON1(9,A);
END PRINT;

READ: PROC(A);
    DCL A ADDRESS;
    CALL MON1(10,A);
END READ;

PRINT$ERROR: PROC (CODE);
    DCL CODE ADDRESS, I BYTE, TEN LIT '39H';
    CALL CRLF;
    CALL PRINT$CHAR(HIGH(CODE));
    CALL PRINT$CHAR(LOW(CODE));
    I = 4;
    DO WHILE (ERROR$CTR(I) := ERROR$CTR(I) + 1) = TEN;
        ERROR$CTR(I) = '0';
        IF I > 0 THEN
```

```
            IF ERROR$CTR(I := I - 1) = ' ' THEN
                ERROR$CTR(I) = '0';
    END;
END PRINT$ERROR;

FATAL$ERROR: PROC(CODE);
    DCL CODE ADDRESS;
    CALL PRINT$ERROR(CODE);
    CALL MON1(9,.(' FATAL ERROR$'));
    CALL BOOTER;
END FATAL$ERROR;

SET$DMA: PROC;
    CALL MON1 (26, CURRENT$FCB + START$OFFSET);
END SET$DMA;

OPEN: PROC (ADDR) BYTE;
    DCL ADDR ADDRESS, RET BYTE;
    CALL MON1 (26,80H);
    RET = MON2(15,ADDR);
    CALL SET$DMA;  /* RESET BUFFER */
    RETURN RET;
END OPEN;

CLOSE: PROC (ADDR);
    DCL ADDR ADDRESS;
    CALL MON1 (26,80H);
    IF MON2(16,ADDR) = 255 THEN CALL FATAL$ERROR('CL');
    CALL SET$DMA;  /* RESET BUFFER */
END CLOSE;

DELETE: PROC;
    CALL MON1(19,CURRENT$FCB);
END DELETE;

MAKE: PROC (ADDR);
    DCL ADDR ADDRESS;
    IF MON2(22,ADDR) = 255 THEN CALL FATAL$ERROR('ME');
END MAKE;

DISK$READ: PROC BYTE;
    RETURN MON2(20,CURRENT$FCB);
END DISK$READ;

DISK$WRITE: PROC BYTE;
    RETURN MON2(21,CURRENT$FCB);
END DISK$WRITE;

/* * * * * * * * * * UTILITY PROCEDURES * * * * * * * * * * */

DCL
```

```
SUBSCRIPT          (8)       ADDRESS;

RES: PROC(ADDR) ADDRESS;
    /* THIS PROC RESOLVES THE ADDRESS OF A SUBSCRIPTED
    IDENTIFIER OR A LITERAL CONSTANT */
    DCL ADDR ADDRESS,
        I BYTE;
    IF ADDR > 32 THEN
        IF ADDR > HI$FREE$MEM THEN RETURN ADDR - HI$OFFSET;
        ELSE RETURN ADDR + LOW$OFFSET;
    IF ADDR < 8 THEN RETURN SUBSCRIPT(ADDR);
    IF ADDR > 12 THEN RETURN CALL$PTR(ADDR - 12);
    DO CASE ADDR - 10;
        RETURN .(' ');
        RETURN .(27H);
        RETURN .('0');
    END;
    RETURN 0;
END RES;

MOVE: PROC(FROM,DESTINATION,COUNT);
    DCL (FROM,DESTINATION,COUNT) ADDRESS,
        (F BASED FROM, D BASED DESTINATION) BYTE;
    DO WHILE (COUNT := COUNT - 1) <> 0FFFFH;
        D = F;
        FROM = FROM + 1;
        DESTINATION = DESTINATION + 1;
    END;
END MOVE;

FILL: PROC(DESTINATION,COUNT,CHAR);
    DCL (DESTINATION,COUNT) ADDRESS,
        (CHAR,D BASED DESTINATION) BYTE;
    DO WHILE (COUNT := COUNT - 1) <> 0FFFFH;
        D = CHAR;
        DESTINATION = DESTINATION + 1;
    END;
END FILL;

FILLER: PROC BYTE;
    IF C$ADDR(1) = 0BH THEN RETURN 27H;
    ELSE IF C$ADDR(1) = 0CH THEN RETURN '0';
    ELSE RETURN ' ';
END FILLER;

CONVERT$TO$HEX: PROC(POINTER,COUNT) ADDRESS;
    DCL POINTER ADDRESS, (COUNT,CHAR,CTR) BYTE;
    A$CTR = 0;
    BASE = POINTER;
    DO CTR = 0 TO COUNT - 1;
        IF ((CHAR := B$BYTE(CTR)) = '-') OR
```

```
            ((CHAR - ZONE >= '0') AND
             (CHAR - ZONE <= '9')) THEN RETURN A$CTR := 0;
        IF  CHAR = '.' THEN RETURN A$CTR;
        IF  CHAR <> '+' THEN
            A$CTR = SHL(A$CTR,3) + SHL(A$CTR.1) +
                (CHAR - '0');
    END;
    RETURN A$CTR;
END CONVERT$TO$HEX;

/* * * * * * * * * CODE CONTROL PROCEDURES * * * * * * * * * */

DCL BRANCH$FLAG         BYTE      INITIAL(FALSE);

INC$PTR: PROC (COUNT);
    DCL COUNT BYTE;
    PROGRAM$COUNTER = PROGRAM$COUNTER + COUNT;
END INC$PTR;

GET$OP$CODE: PROC BYTE;
    CTR = C$BYTE(0);
    CALL INC$PTR(1);
    RETURN CTR;
END GET$OP$CODE;

COND$BRANCH: PROC(COUNT);
    /* THIS PROC CONTROLS BRANCHING INSTRUCTIONS */
    DCL COUNT BYTE;
    IF  BRANCH$FLAG THEN
        DO;
            BRANCH$FLAG = FALSE;
            PROGRAM$COUNTER = C$ADDR(COUNT);
        END;
    ELSE CALL INC$PTR(SHL(COUNT,1) + 2);
END COND$BRANCH;

INCR$OR$BRANCH: PROC(MARK);
    DCL MARK BYTE;
    IF MARK THEN CALL INC$PTR(2);
    ELSE PROGRAM$COUNTER = C$ADDR(0);
END INCR$OR$BRANCH;

/* * * * * * * * * * * * * COMPARISONS * * * * * * * * * * * * */

CHAR$COMPARE: PROC BYTE;
    DCL A$ADDR ADDRESS;
    A$ADDR = FILLER;
    IF C$ADDR(1) > 09H AND C$ADDR(1) < 0DH THEN
        DO A$CTR = 0 TO C$ADDR(2) - 1;
            IF B$BYTE(A$CTR) > A$ADDR THEN RETURN 1;
            IF B$BYTE(A$CTR) < A$ADDR THEN RETURN 0;
```

```
        END;
    ELSE
        DO A$CTR = 0 TO C$ADDR(2) - 1;
            IF B$BYTE(A$CTR) > H$BYTE(A$CTR) THEN RETURN 1;
            IF B$BYTE(A$CTR) < H$BYTE(A$CTR) THEN RETURN 0;
        END;
    RETURN 2;
END CHAR$COMPARE;

NUMERIC: PROC(CHAR) BYTE;
    DCL CHAR BYTE;
    RETURN (CHAR >= '0') AND (CHAR <= '9');
END NUMERIC;

LETTER: PROC(CHAR) BYTE;
    DCL CHAR BYTE;
    RETURN (CHAR >= 'A') AND (CHAR <= 'Z');
END LETTER;

SIGN: PROC(CHAR) BYTE;
    DCL CHAR BYTE;
    RETURN (CHAR = '+') OR (CHAR = '-');
END SIGN;

CHK$S$NUM: PROC(BASE) BYTE;
    DCL BASE ADDRESS,
            B$BYTE BASED BASE (1) BYTE,
            (I,LENGTH) BYTE;
    DO I = 1 TO (LENGTH := C$ADDR(2) - 1) - 1;
        IF NOT NUMERIC(B$BYTE(I)) THEN RETURN FALSE;
    END;
    IF NUMERIC(B$BYTE(0)) AND NUMERIC(B$BYTE(LENGTH)) THEN
        RETURN FALSE;
    CALL MOVE(BASE,.R0,LENGTH + 1);
    IF NUMERIC(B$BYTE(0) - ZONE) AND
        NUMERIC(B$BYTE(LENGTH)) THEN
            R0(0) = R0(0) - ZONE;
    ELSE IF NUMERIC(B$BYTE(0)) AND
        NUMERIC(B$BYTE(LENGTH) - ZONE) THEN
            R0(LENGTH) = R0(LENGTH) - ZONE;
    ELSE RETURN FALSE;
    RETURN TRUE;
END CHK$S$NUM;

STRING$COMPARE: PROC(PIVOT);
    DCL PIVOT BYTE;
    HOLD = RES(C$ADDR(1));
    IF CHK$S$NUM(BASE := RES(C$ADDR(0))) THEN BASE = .R0;
    ELSE IF CHK$S$NUM(HOLD) THEN HOLD = .R0;
    IF CHAR$COMPARE = PIVOT THEN
        BRANCH$FLAG = NOT BRANCH$FLAG;
```

```
    CALL COND$BRANCH(3);
END STRING$COMPARE;

COMP$NUM$UNSIGNED: PROC;
    BASE = RES(C$ADDR(Ø));
    DO A$CTR = Ø TO C$ADDR(1) - 1;
        IF NOT NUMERIC(B$BYTE(A$CTR)) THEN
            A$CTR = C$ADDR(1) + 1;
    END;
    IF A$CTR = C$ADDR(1) TEEN BRANCH$FLAG = NOT BRANCH$FLAG;
    CALL COND$BRANCH(2);
END COMP$NUM$UNSIGNED;

COMP$NUM$SIGN: PROC;
    DCL (CHAR,SIGN$FLAG) BYTE;
    SIGN$FLAG = FALSE;
    BASE = RES(C$ADDR(Ø));
    DO A$CTR = Ø TO C$ADDR(1) - 1;
        IF NOT NUMERIC(CHAR := B$BYTE(A$CTR)) THEN
          IF (A$CTR = Ø) OR (A$CTR = C$ADDR(1) - 1) THEN
            IF (SIGN(CHAR) OR NUMERIC(CHAR-ZONE)) AND
                NOT SIGN$FLAG THEN
                SIGN$FLAG = TRUE;
            ELSE A$CTR = C$ADDR(1) + 1;
          ELSE A$CTR = C$ALDR(1) + 1;
    END;
    IF A$CTR = C$ADDR(1) THEN BRANCH$FLAG = NOT BRANCH$FLAG;
    CALL COND$BRANCH(2);
END COMP$NUM$SIGN;

COMP$ALPHA: PROC;
    BASE = RES(C$ADDR(Ø));
    DO A$CTR = Ø TO C$ADDR(1) - 1;
        IF NOT LETTER(B$BYTE(A$CTR)) THEN
            A$CTR = C$ADDR(1) + 1;
    END;
    IF A$CTR = C$ADDR(1) THEN BRANCH$FLAG = NOT BRANCH$FLAG;
    CALL COND$BRANCH(2);
END COMP$ALPHA;

/* * * * * * * * * * *NUMERIC OPERATIONS * * * * * * * * * * */

DCL     (RØ,R1,R2) (18)     BYTE, /* REGISTERS */
        DEC$PTØ     BYTE,
        DEC$PT1     BYTE,
        DEC$PT2     BYTE,
        DEC$PTA(3) BYTE     AT(.DEC$PTØ),
        MOVE$FLAG  BYTE     INITIAL(FALSE),
        OVERFLOW   BYTE,
        R$PTR      BYTE,
        REG$LENGTH BYTE     INITIAL(1Ø),
```

```
        SIGN0(3)    BYTE,
        SWITCH      BYTE,
        TEMP        BYTE,
        NEGITIVE    LIT         '0',
        POSITIVE    LIT         '1';

CHECK$FOR$SIGN: PROC(CHAR) BYTE;
    DCL CHAR BYTE;
    IF NUMERIC(CHAR) THEN RETURN POSITIVE;
    IF NUMERIC(CHAR - ZONE) THEN RETURN NEGITIVE;
    CALL PRINT$ERROR('SI');
    RETURN POSITIVE;
END CHECK$FOR$SIGN;

STORE$IMMEDIATE: PROC;
    DO CTR = 0 TO 9;
         R0(CTR) = R2(CTR);
    END;
    DEC$PT0 = DEC$PT2;
    SIGN0(0) = SIGN0(2);
END STORE$IMMEDIATE;

ONE$LEFT: PROC;
    DCL CTR BYTE;
    IF SHL(B$BYTE(0),4) = 0 OR MOVE$FLAG THEN
      DO;
        DO CTR = 0 TO REG$LENGTH - 2;
             B$BYTE(CTR) = SHL(B$BYTE(CTR),4) OR
                           SHR(B$BYTE(CTR + 1),4);
        END;
        B$BYTE(REG$LENGTH - 1) =
            SHL(B$BYTE(REG$LENGTH - 1),4);
      END;
    ELSE OVERFLOW = TRUE;
END ONE$LEFT;

ONE$RIGHT: PROC;
    DCL CTR BYTE;
    CTR = REG$LENGTH;
    DO INDEX = 1 TO REG$LENGTH - 1;
        CTR = CTR - 1;
        B$BYTE(CTR) = SHR(B$BYTE(CTR),4) OR
                      SHL(B$BYTE(CTR - 1),4);
    END;
    B$BYTE(0) = SHR(B$BYTE(0),4);
    IF B$BYTE(0) = 09H THEN
        B$BYTE(0) = 99H;
END ONE$RIGHT;

SHIFT$RIGHT: PROC(COUNT);
    DCL COUNT BYTE;
```

```
    DO CTR = 1 TO COUNT;
        CALL ONE$RIGHT;
    END;
END SHIFT$RIGHT;

SHIFT$LEFT: PROC (COUNT);
    DCL COUNT BYTE;
    OVERFLOW = FALSE;
    IF COUNT = 0 THEN
        DO;
            CTR = 0;
            RETURN;
        END;
    DO CTR = 0 TO COUNT - 1;
         CALL ONE$LEFT;
         IF OVERFLOW AND NOT MOVE$FLAG THEN RETURN;
    END;
END SHIFT$LEFT;

ALLIGN: PROC;
    DCL (X,Y) BYTE;
    RIGHT$OP:PROC(ADDR);
        DCL ADDR ADDRESS;
        IF OVERFLOW THEN
           DO;
             BASE = ADDR;
             CALL SHIFT$RIGHT(Y := X - CTR);
             OVERFLOW = FALSE;
           END;
    END RIGHT$OP;

    Y = 0;
    IF DEC$PT0>DEC$PT1 THEN
        DO;
            BASE = .R1;
            CALL SHIFT$LEFT(X := DEC$PT0 - DEC$PT1);
            DEC$PT1 = DEC$PT1+CTR;
            CALL RIGHT$OP(.R0);
            DEC$PT0 = DEC$PT0 - Y;
        END;
    ELSE
        DO;
            BASE = .R0;
            CALL SHIFT$LEFT(X := DEC$PT1 - DEC$PT0);
            DEC$PT0 = DEC$PT0+CTR;
            CALL RIGHT$OP(.R1);
            DEC$PT1 = DEC$PT1 - Y;
        END;
END ALLIGN;

ADD$TO$END: PROC(CY);
```

```
    DCL (CY,I,J) BYTE;
    CTR = REG$LENGTH - 1;
    DO J = 1 TO REG$LENGTH;
        I = B$BYTE(CTR);
        I = DEC(I+CY);
        CY = CARRY AND 1;
        B$BYTE(CTR) = I;
        CTR = CTR - 1;
    END;
END ADD$TO$END;

ADD$R0: PROC(SECOND, DEST);
    DCL (SECOND, DEST) ADDRESS, (CY,A,B,I,J) BYTE;
    HOLD = SECOND;
    BASE = DEST;
    CY = 0;
    CTR = REG$LENGTH - 1;
    DO J = 1 TO REG$LENGTH;
         A = R0(CTR);
         B = H$BYTE(CTR);
         I = DEC(A+CY);
         CY = CARRY;
         I = DEC(I + B);
         CY = (CY OR CARRY) AND 1;
         B$BYTE(CTR) = I;
         CTR = CTR - 1;
    END;
    IF CY THEN CALL ADD$TO$END(CY);
END ADD$R0;

COMPLIMENT: PROC(NUMB);
    DCL NUMB BYTE;
    SIGN0(NUMB) = SIGN0(NUMB) XOR 1;  /* COMPLIMENT SIGN */
    DO CASE NUMB;
         HOLD = .R0;
         HOLD = .R1;
         HOLD = .R2;
    END;
    DO CTR = 0 TO REG$LENGTH - 1;
         H$BYTE(CTR) = 99H - H$BYTE(CTR);
    END;
END COMPLIMENT;

R2$ZERO: PROC BYTE;
   DCL I BYTE;
   IF (SHL(R2(0),4) <> 0) OR (SHR(R2(9),4) <> 0)
   THEN RETURN FALSE;
   ELSE DO I = 1 TO 8;
          IF R2(I) <> 0 THEN RETURN FALSE;
        END;
   RETURN TRUE;
```

```
END R2$ZERO;

LEADING$ZEROES: PROC(ADDR) BYTE;
    DCL COUNT BYTE, ADDR ADDRESS;
    COUNT = 0;
    BASE = ADDR;
    DO CTR = 0 TO 9;
        IF (B$BYTE(CTR) AND 0F0H) <> 0 THEN RETURN COUNT;
        COUNT = COUNT + 1;
        IF (B$BYTE(CTR) AND 0FH) <> 0 THEN RETURN COUNT;
        COUNT = COUNT + 1;
    END;
    RETURN COUNT;
END LEADING$ZEROES;

CHECK$RESULT: PROC;
    IF SHR(R2(0),4) = 9 THEN CALL COMPLIMENT(2);
    BASE = .R2;
    CALL ADD$TO$END(05H);
    IF (SHR(R2(0),4)<>0) AND (DEC$PT2 = 0) THEN
        OVERFLOW = TRUE;
    ELSE
        IF (SHR(R2(0),4) <> 0) THEN
            DO;
                CALL SHIFT$RIGHT(1);
                DEC$PT2 = DEC$PT2 - 1;
            END;
    B$BYTE(9) = B$BYTE(9) AND 0F0H;
    IF LEADING$ZEROES(.R2) > 19 THEN
        SIGN0(2) = POSITIVE;
END CHECK$RESULT;

CHECK$SIGN: PROC;
    SIGN0(2) = POSITIVE;
    IF SIGN0(0) AND SIGN0(1) THEN RETURN;
    IF (NOT SIGN0(0)) AND (NOT SIGN0(1)) THEN
        DO;
            SIGN0(2) = NEGITIVE;
            RETURN;
        END;
    IF SIGN0(0) THEN CALL COMPLIMENT(1);
    ELSE CALL COMPLIMENT(0);
END CHECK$SIGN;

CHECK$NUMERIC: PROC;
    DCL I BYTE;
    BASE = .R0;
    DO I = 0 TO 27;
        IF NOT NUMERIC(SHR(B$BYTE(I),4) OR '0') OR
           NOT NUMERIC((B$BYTE(I) AND 0FH) OR '0') THEN
               CALL PRINT$ERROR('NE');
```

```
    END;
END CHECK$NUMERIC;

CHECK$DECIMAL: PROC;
    IF DEC$PT2<>(CTR := C$BYTE(3)) THEN
        DO;
            MOVE$FLAG = TRUE;
            BASE = .R2;
            IF DEC$PT2 > CTR THEN CALL
                SHIFT$RIGHT(DEC$PT2 - CTR);
            ELSE CALL SHIFT$LEFT(CTR-DEC$PT2);
            MOVE$FLAG = FALSE;
        END;
    IF LEADING$ZEROES(.R2) < 19 - C$BYTE(2) THEN
        OVERFLOW = TRUE;
END CHECK$DECIMAL;

ADD: PROC;
    CALL CHECK$NUMERIC;
    OVERFLOW = FALSE;
    CALL ALLIGN;
    CALL CHECK$SIGN;
    DEC$PT2 = DEC$PT0;
    CALL ADDR0(.R1,.R2);
    CALL CHECK$RESULT;
END ADD;

ADD$SERIES: PROC(COUNT);
    DCL (I,COUNT) BYTE;
    DO I = 1 TO COUNT;
        CALL ADD$R0(.R2,.R2);
    END;
END ADD$SERIES;

SET$MULT$DIV: PROC;
    CALL CHECK$NUMERIC;
    OVERFLOW = FALSE;
    REG$LENGTH = 18;
    SIGN0(2) = (NOT (SIGN0(0) XOR SIGN0(1))) AND 01H;
    CALL FILL(.R2,18,0);
END SET$MULT$DIV;

R1$GREATER: PROC BYTE;
    DCL I BYTE;
    DO CTR = 0 TO 9;
        IF R1(CTR)>(I := 99H - R0(CTR)) THEN RETURN TRUE;
        IF R1(CTR)<I THEN RETURN FALSE;
    END;
    RETURN TRUE;
END R1$GREATER;
```

```
MULTIPLY: PROC(VALUE);
    DCL VALUE BYTE;
    IF VALUE<>Ø THEN CALL ADD$SERIES(VALUE);
    BASE = .RØ;
    CALL ONE$LEFT;
END MULTIPLY;

DIVIDE: PROC;
    DCL (I, J, K, X) BYTE;
    IF LEADING$ZEROES(.RØ) > 19 THEN
        DO;
            OVERFLOW = TRUE;
            RETURN;
        END;
    IF LEADING$ZEROES(.R1) > 19 THEN
        DO;
            CALL FILL(.R2,18,Ø);
            RETURN;
        END;
    CALL SET$MULT$DIV;
    BASE = .RØ;
    CALL SHIFT$LEFT(17);
    DEC$PTØ = DEC$PTØ + CTR;
    BASE = .R1;
    CALL SHIFT$LEFT(17);
    DEC$PT1 = DEC$PT1 + CTR;
    OVERFLOW = FALSE;
    IF DEC$PTØ > 17 THEN
        IF DEC$PT1 < (X := DEC$PTØ - 17) THEN
            DO;
                OVERFLOW = TRUE;
                DEC$PT2 = Ø;
            END;
        ELSE
            DEC$PT2 = DEC$PT1 - X;
    ELSE
        DEC$PT2 = DEC$PT1 + (17 - DEC$PTØ);
    CALL COMPLIMENT(Ø);
    DO I = 1 TO 19;
         J = Ø;
         DO WHILE R1$GREATER;
              CALL ADD$RØ(.R1,.R1);
              IF R1(Ø) = 99H THEN
                       CALL COMPLIMENT (1);
              J = J + 1;
         END;
         K = SHR(I,1);
         IF I THEN R2(K) = R2(K) OR J;
         ELSE R2(K) = R2(K) OR SHL(J,4);
         BASE = .RØ;
         CALL ONE$RIGHT;
```

```
    END;
    REG$LENGTH = 10;
    CALL CHECK$RESULT;
END DIVIDE;

LOAD$A$CHAR: PROC(CHAR);
    DCL CHAR BYTE;
    IF (SWITCH := NOT SWITCH) THEN
        B$BYTE(R$PTR) = B$BYTE(R$PTR) OR SHL(CHAR - 30H,4);
    ELSE B$BYTE(R$PTR := R$PTR-1) = CHAR - 30H;
END LOAD$A$CHAR;

LOAD$NUMBERS: PROC(ADDR,CNT);
    DCL ADDR ADDRESS, (I,CNT)BYTE;
    HOLD = RES(ADDR);
    CTR = CNT;
    DO INDEX = 1 TO CNT;
        CTR = CTR - 1;
        CALL LOAD$A$CHAR(H$BYTE(CTR));
    END;
    CALL INC$PTR(5);
END LOAD$NUMBERS;

SET$LOAD: PROC (SIGN$IN);
  DCL (CTR,SIGN$IN) BYTE;
    DO CASE (CTR := C$BYTE(4));
        BASE = .R0;
        BASE = .R1;
        BASE = .R2;
    END;
    DEC$PTA(CTR) = C$BYTE(3);
    SIGN0(CTR) = SIGN$IN;
    CALL FILL (BASE,18,0);
    R$PTR = 9;
    SWITCH = FALSE;
END SET$LOAD;

LOAD$NUMERIC: PROC;
    CALL SET$LOAD(1);
    CALL LOAD$NUMBERS(RES(C$ADDR(0)),C$BYTE(2));
END LOAD$NUMERIC;

LOAD$NUM$LIT: PROC;
    DCL(LIT$SIZE,FLAG) BYTE;

    CHAR$SIGN: PROC;
        LIT$SIZE = LIT$SIZE - 1;
        HOLD = HOLD + 1;
    END CHAR$SIGN;

    LIT$SIZE = C$BYTE(2);
```

```
    HOLD = RES(C$ADDR(0));
    IF H$BYTE(0) = '-' THEN
        DO;
            CALL CHAR$SIGN;
            CALL SET$LOAD(NEGITIVE);
        END;
    ELSE
        DO;
            IF H$BYTE(0) = '+' THEN CALL CHAR$SIGN;
            CALL SET$LOAD(POSITIVE);
        END;
    FLAG = 0;
    CTR = LIT$SIZE;
    DO INDEX = 1 TO LIT$SIZE;
         CTR = CTR - 1;
         IF H$BYTE(CTR) = '.' THEN FLAG=LIT$SIZE - (CTR + 1);
         ELSE CALL LOAD$A$CHAR(H$BYTE(CTR));
    END;
    DEC$PTA(C$BYTE(4)) = FLAG;
    CALL INC$PTR(5);
END LOAD$NUM$LIT;

STORE$ONE: PROC;
    IF(SWITCH := NOT SWITCH) THEN
        B$BYTE(0) = SHR(H$BYTE(0),4) OR '0';
    ELSE
        DO;
             HOLD = HOLD - 1;
            B$BYTE(0) = (H$BYTE(0) AND 0FH) OR '0';
        END;
    BASE = BASE - 1;
END STORE$ONE;

STORE$AS$CHAR: PROC(COUNT);
    DCL COUNT BYTE;
    SWITCH = FALSE;
    HOLD = .R2 + 9;
    IF C$BYTE(4) <> SER OR NOT OVERFLOW THEN
        DO CTR = 1 TO COUNT;
            CALL STORE$ONE;
        END;
END STORE$AS$CHAR;

SET$ZONE: PROC (ADDR);
    DCL ADDR ADDRESS;
    IF NOT SIGN0(2) THEN
        DO;
            BASE = ADDR;
            IF C$BYTE(4) <> SER OR NOT OVERFLOW THEN
                B$BYTE(0) = B$BYTE(0) + ZONE;
        END;
```

```
    CALL INC$PTR(4);
END SET$ZONE;

SET$SIGN$SEP: PROC (ADDR);
    DCL ADDR ADDRESS;
    BASE = ADDR;
    IF C$BYTE(4) <> SER OR NOT OVERFLOW THEN
        IF SIGN0(2) THEN B$BYTE(0) = '+';
        ELSE B$BYTE(0) = '-';
    CALL INC$PTR(4);
END SET$SIGN$SEP;

STORE$NUMERIC: PROC;
    CALL CHECK$DECIMAL;
    BASE = RES(C$ADDR(0)) + C$BYTE(2) - 1;
    CALL STORE$AS$CHAR(C$BYTE(2));
END STORE$NUMERIC;

MOVE$NUM$EDITED: PROC;
DCL     CHAR            BYTE,
        COUNT           BYTE,
        FLAG(2)         BYTE,
        FLOAT$VALUE     BYTE,
        LAST$LOAD       BYTE,
        LENGTH          BYTE,
        MAX$LOAD$PT     BYTE,
        MIN$LOAD$PT     BYTE,
        PSIT$DEC        BYTE,
        PSIT$SIGN       BYTE,
        SIGN$OUT        BYTE;

    FLOAT$CHECK: PROC(INDEX);
        DCL INDEX BYTE;
        IF FLAG(INDEX) THEN
            FLOAT$VALUE = CHAR;
        ELSE
            DO;
                FLAG(INDEX) = TRUE;
                IF CTR <> MAX$LOAD$PT OR INDEX = 0 THEN
                    MIN$LOAD$PT = CTR + 1;
                IF INDEX = 1 THEN
                    PSIT$SIGN = CTR;
            END;
    END FLOAT$CHECK;

    FLOAT$VALUE,MIN$LOAD$PT = 0;
    FLAG(0),FLAG(1) = FALSE;
    PSIT$DEC = C$BYTE(11);
    PSIT$SIGN = C$BYTE(8);
    MAX$LOAD$PT = C$BYTE(8) - 1;
    HOLD = RES(C$ADDR(0));
```

```
CALL MOVE(RES(C$ADDR(3)),HOLD,C$ADDR(4));
IF H$BYTE(MAX$LOAD$PT) = 'B' OR
   H$BYTE(MAX$LOAD$PT) = 'R' THEN
    DO;
        MAX$LOAD$PT = MAX$LOAD$PT - 2;
        PSIT$DEC = PSIT$DEC - 2;
        PSIT$SIGN = PSIT$SIGN - 2;
    END;
DO CTR = 0 TO MAX$LOAD$PT;
    CHAR = H$BYTE(CTR);
    IF CHAR = '9' THEN
        H$BYTE(CTR) = '0';
    ELSE IF CHAR = '$' THEN
        CALL FLOAT$CHECK(0);
    ELSE IF SIGN(CHAR) THEN
        CALL FLOAT$CHECK(1);
    ELSE IF CHAR = 'Z' THEN
        FLOAT$VALUE = CHAR;
    ELSE IF CHAR = 'B' THEN
        H$BYTE(CTR) = ' ';
    IF CTR > MAX$LOAD$PT - PSIT$DEC THEN
        IF CHAR = '/' OR CHAR = ' ' OR
           CHAR = '0' OR CHAR = ',' THEN
               PSIT$DEC = PSIT$DEC - 1;
END; /* DO CTR = 0 TO MAX$LOAD$PT */
IF PSIT$SIGN = MAX$LOAD$PT THEN
    DO;
        MAX$LOAD$PT = MAX$LOAD$PT - 1;
        PSIT$DEC = PSIT$DEC - 1;
    END;
LENGTH = C$ADDR(2);
BASE = .R0;
CALL FILL(BASE,36,'0');
CALL MOVE(RES(C$ADDR(1)),BASE,LENGTH);
IF SIGN(B$BYTE(0)) THEN /* CHECK FOR LEADING SIGN */
    DO;
        SIGN$OUT = B$BYTE(0);
        BASE = BASE + 1;
        LENGTH = LENGTH - 1;
    END;
ELSE IF SIGN(B$BYTE(C$BYTE(4) - 1)) THEN
    DO;
        SIGN$OUT = B$BYTE(C$BYTE(4) - 1);
        LENGTH = LENGTH - 1;
    END;
ELSE IF NOT CHECK$FOR$SIGN(B$BYTE(C$BYTE(4) - 1)) THEN
    DO; /* CHECK FOR TRAILING IMBEDDED SIGN */
        SIGN$OUT = '-';
        B$BYTE(C$BYTE(4) - 1) = B$BYTE(C$BYTE(4) - 1)
            - ZONE;
    END;
```

```
ELSE IF NOT CHECK$FOR$SIGN(B$BYTE(0)) THEN
    DO; /* CHECK FOR LEADING IMBEDDED SIGN */
        SIGN$OUT = '-';
        B$BYTE(0) = B$BYTE(0) - ZONE;
    END;
ELSE SIGN$OUT = '+';
IF PSIT$DEC <> C$BYTE(10) THEN
    DO; /* ALIGN DECIMAL POSITIONS */
        IF PSIT$DEC < C$BYTE(10) THEN
            LENGTH = LENGTH - (C$BYTE(10) - PSIT$DEC);
        ELSE
            LENGTH = LENGTH + (PSIT$DEC - C$BYTE(10));
    END;
CTR = LENGTH - 1;
COUNT,LAST$LOAD = MAX$LOAD$PT;
DO INDEX = 1 TO LENGTH;
    DO WHILE (H$BYTE(COUNT) = ' ' OR H$BYTE(COUNT) = '0'
           OR H$BYTE(COUNT) = '/' OR H$BYTE(COUNT) = '.'
           OR H$BYTE(COUNT) = ',') AND
           (COUNT <= MAX$LOAD$PT);
              COUNT = COUNT - 1;
    END;
    IF B$BYTE(CTR) <> '.' THEN
        DO;
            IF B$BYTE(CTR) <> '0' THEN
                IF (COUNT < MIN$LOAD$PT) OR
                   (COUNT = 255) THEN
                    INDEX = LENGTH;
                ELSE
                    DO;
                        H$BYTE(COUNT) = B$BYTE(CTR);
                        LAST$LOAD = COUNT;
                    END;
            COUNT = COUNT - 1;
        END;
    CTR = CTR - 1;
END;
IF FLOAT$VALUE <> 0 THEN
    DO;
        CTR = 0;
        DO WHILE H$BYTE(CTR) <> FLOAT$VALUE;
            CTR = CTR + 1;
        END;
        DO WHILE (H$BYTE(CTR) = ',' OR H$BYTE(CTR) = '0'
               OR H$BYTE(CTR) = ' ' OR H$BYTE(CTR) = '/'
               OR H$BYTE(CTR) = FLOAT$VALUE)
               AND (CTR <= MAX$LOAD$PT);
            H$BYTE(CTR) = ' ';
            CTR = CTR + 1;
        END;
        IF FLOAT$VALUE <> 'Z' THEN
```

```
                DO;
                    H$BYTE(CTR := CTR - 1) = FLOAT$VALUE;
                    IF SIGN(FLOAT$VALUE) THEN
                        PSIT$SIGN = CTR;
                END;
        END;
    DO CTR = 0 TO LAST$LOAD;
        IF H$BYTE(CTR) = '0' THEN
            H$BYTE(CTR) = '0';
        ELSE
            IF H$BYTE(CTR) = ',' AND
               H$BYTE(CTR - 1) = '*' THEN
                 H$BYTE(CTR) = '*';
    END;
    DO CTR = LAST$LOAD + 1 TO MAX$LOAD$PT;
        IF H$BYTE(CTR) = '*' OR H$BYTE(CTR) = '$' OR
           SIGN(H$BYTE(CTR)) OR H$BYTE(CTR) = '0' THEN
               H$BYTE(CTR) = '0';
    END;
    IF PSIT$SIGN < C$BYTE(8) THEN
        IF E$BYTE(PSIT$SIGN) = '+' THEN
            H$BYTE(PSIT$SIGN) = SIGN$OUT;
        ELSE
            IF SIGN$OUT = '+' THEN
                DO;
                    IF H$BYTE(PSIT$SIGN) <> '-' THEN
                        H$BYTE(PSIT$SIGN + 1) = ' ';
                    H$BYTE(PSIT$SIGN) = ' ';
                END;
    CALL INC$PTR(12);
END MOVE$NUM$EDITED;

/* * * * * * * * * * INPUT-OUTPUT ACTIONS * * * * * * * * * * */

DCL BUFF$PTR            ADDRESS,
    BUFF$BYTE           BASED     BUFF$PTR (1)    BYTE,
    BUFF$END            ADDRESS,
    BUFF$LENGTH         LIT       '128',
    BUFFSTART           ADDRESS,
    CHAR                BYTE,
    CON$BUFF            ADDRESS   INITIAL (80H),
    CON$BYTE            BASED     CON$BUFF        BYTE,
    CON$INPUT           ADDRESS   INITIAL (82H),
    CONTROL$FLAG        BYTE      INITIAL (FALSE),
    CURRENT$FLAG        BYTE,
    EOF$FLAG$OFFSET     LIT       '36',
    EXTENT$OFFSET       LIT       '12',
    FCB$ADDR$A          BASED     CURRENT$FCB (1) ADDRESS,
    FCB$BYTE$A          BASED     CURRENT$FCB (1) BYTE,
    FLAG$OFFSET         LIT       '33',
    HIGH$VALUE          LIT       '0FFH',
```

```
    INVALID             BYTE,
    PAG                 LIT         '22', /* CODE FOR PAGE */
    PTR$OFFSET          LIT         '17',
    REC$NO              LIT         '32',
    REWRITE$FLAG        BYTE        INITIAL(0H),
    TERMINATOR          LIT         '1AH',
    TOP$OF$PAGE         LIT         '0CH',
    VAR$END             LIT         'CR',
    WTF                 LIT         '48'; /* CODE FOR WRITE */

ACCEPT: PROC;
    CALL CRLF;
    CALL PRINT$CHAR(3FH);
    CALL FILL(CON$INPUT,C$BYTE(2),' ');
    CON$BYTE = 128;
    CALL READ(CON$BUFF);
    CALL MOVE(CON$INPUT,RES(C$ADDR(0)),C$BYTE(2));
    CALL INC$PTR(3);
END ACCEPT;

DISPLAY: PROC;
    DCL  B$CNT BYTE;
    BASE = RES(C$ADDR(0));
    IF NOT C$BYTE(3) THEN CALL CRLF;
    B$CNT = C$BYTE(2);
    DO CTR = 0 TO B$CNT - 1;
        CALL PRINT$CHAR(B$BYTE(CTR));
    END;
    CALL INC$PTR(4);
END DISPLAY;

GET$FILE$TYPE: PROC BYTE;
    BASE = C$ADDR(0);
    RETURN B$BYTE(FLAG$OFFSET);
END GET$FILE$TYPE;

SET$FILE$TYPE: PROC(TYPE);
    DCL TYPE BYTE;
    BASE = C$ADDR(0);
    IF GET$FILE$TYPE <> 0 THEN CALL FATAL$ERROR('0E');
    B$BYTE(FLAG$OFFSET) = TYPE;
END SET$FILE$TYPE;

SET$I$O: PROC;
    INVALID = FALSE;
    IF C$ADDR(0) = CURRENT$FCB THEN RETURN;
    /* STORE CURRENT POINTERS AND SET INTERNAL WRITE MARK */
    BASE = CURRENT$FCB;
    FCB$ADDR$A(PTR$OFFSET) = BUFF$PTR;
    FCB$BYTE$A(FLAG$OFFSET) = CURRENT$FLAG;
    /* LOAD NEW VALUES */
```

```
    BUFF$END = (BUFF$START := (CURRENT$FCB := C$ADDR(0)) +
        START$OFFSET) + BUFF$LENGTH;
    CURRENT$FLAG = FCB$BYTE$A(FLAG$OFFSET);
    BUFF$PTR = FCB$ADDR$A(PTR$OFFSET);
END SET$I$O;

OPEN$FILE: PROC(TYPE);
    DCL TYPE BYTE;
    CALL SET$FILE$TYPE(TYPE);
    CURRENT$FCB = C$ADDR(0);
    FCB$BYTE$A(EXTENT$OFFSET) = 0;
    CTR = OPEN(CURRENT$FCB);
    DO CASE TYPE - 1;
        /* INPUT */
        DO;
            IF CTR = 255 THEN CALL FATAL$ERROR('NF');
        END;
        /* OUTPUT */
        DO;
            CALL DELETE;
            CALL MAKE(C$ADDR(0));
        END;
        ; /* CASE 2 NOT USED */
        /* I-O */
        DO;
            IF CTR = 255 THEN CALL FATAL$ERROR('NF');
        END;
    END; /* DO CASE TYPE - 1 */
    FCB$BYTE$A(REC$NO) = 0; /* SET THE RECORD NUMBER IN FCB */
    FCB$BYTE$A(EOF$FLAG$OFFSET) = FALSE; /* SET THE EOF OFF */
    BUFF$END = (BUFF$START := (CURRENT$FCB + START$OFFSET)) +
                BUFF$LENGTH;
    CURRENT$FLAG = FCB$BYTE$A(FLAG$OFFSET);
    BUFF$PTR,FCB$ADDR$A(PTR$OFFSET) = BUFF$START - 1;
    CALL INC$PTR(2);
END OPEN$FILE;

WRITE$MARK: PROC BYTE;
    RETURN ROL(CURRENT$FLAG,1);
END WRITE$MARK;

SET$WRITE$MARK: PROC;
    CURRENT$FLAG = CURRENT$FLAG OR 80H;
END SET$WRITE$MARK;

WRITE$RECORD: PROC;
    CALL SET$DMA;
    CURRENT$FLAG = CURRENT$FLAG AND 0FH;
    IF (CTR := DISK$WRITE) = 0 THEN RETURN;
    CALL PRINT$ERROR('W8');
    INVALID = TRUE;
```

```
END WRITE$RECORD;

READ$RECORD: PROC;
    CALL SET$DMA;
    IF WRITE$MARK THEN CALL WRITE$RECORD;
    IF (CTR := DISK$READ) = 0 THEN RETURN;
    IF CTR = 1 THEN FCB$BYTE$A(EOF$FLAG$OFFSET) = TRUE;
    INVALID = TRUE;
END READ$RECORD;

READ$BYTE: PROC BYTE;
    IF (BUFF$PTR := BUFF$PTR + 1) >= BUFFEND THEN
        DO;
            CALL READ$RECORD;
            IF FCB$BYTE$A(EOF$FLAG$OFFSET) THEN
                RETURN TERMINATOR;
            BUFF$PTR = BUFF$START;
        END;
    RETURN BUFF$BYTE(0);
END READ$BYTE;

WRITE$BYTE: PROC (CHAR);
    DCL CHAR BYTE;
    IF (BUFF$PTR := BUFF$PTR+1) >= BUFF$END THEN
        DO;
            CALL WRITE$RECORD;
            BUFF$PTR = BUFF$START;
            IF REWRITE$FLAG THEN
               DO;
                  CALL READ$RECORD;
                  FCB$BYTE$A(REC$NO) = FCB$BYTE$A(REC$NO) - 1;
               END;
        END;
    CALL SET$WRITE$MARK;
    BUFF$BYTE(0) = CHAR;
END WRITE$BYTE;

WRITE$END$MARK: PROC;
    CALL WRITE$BYTE(CR);
    CALL WRITE$BYTE(LF);
END WRITE$END$MARK;

READ$END$MARK: PROC;
    IF (READ$BYTE<>CR) OR (READ$BYTE<>LF) THEN
        CALL PRINT$ERROR('EM');
END READ$END$MARK;

READ$VARIABLE:PROC;
    CALL SET$I$O;
    BASE = C$ADDR(2);
    CALL FILL(C$ADDR(2),C$ADDR(1),' ');
```

```
    DO A$CTR = Ø TO C$ADDR(1) - 1;
         IF (CTR := READ$BYTE) = VAR$END THEN
             DO;
                 CTR = READ$BYTE;
                 RETURN;
             END;
         IF CTR = TERMINATOR THEN
             DO;
                 FCB$BYTE$A(EOF$FLAG$OFFSET) = TRUE;
                 RETURN;
             END;
         B$BYTE(A$CTR) = CTR;
    END;
    CALL READ$END$MARK;
END READ$VARIABLE;

WRITE$VARIABLE: PROC;
    DCL COUNT ADDRESS;
    CALL SET$I$O;
    BASE = C$ADDR(1);
    COUNT = C$ADDR(2);
    DO WHILE ((B$BYTE(COUNT := COUNT - 1) = ' ')
          AND (COUNT <> Ø));
    END;
    DO A$CTR = Ø TO COUNT;
        CALL WRITE$BYTE(B$BYTE(A$CTR));
    END;
    CALL WRITE$END$MARK;
END WRITE$VARIABLE;

READ$TO$MEMORY: PROC;
    DCL CHAR BYTE;
    BASE = C$ADDR(1);
    DO A$CTR = Ø TO C$ADDR(2) - 1;
        IF (CHAR := READ$BYTE) = TERMINATOR THEN
            DO;
                INVALID,FCB$BYTE$A(EOF$FLAG$OFFSET) = TRUE;
                RETURN;
            END;
        ELSE B$BYTE(A$CTR) = CHAR;
    END;
    CALL READ$END$MARK;
END READ$TO$MEMORY;

WRITE$FROM$MEMORY: PROC;
    BASE = RES(C$ADDR(1));
    DO A$CTR = Ø TO C$ADDR(2) - 1;
        CALL WRITE$BYTE(B$BYTE(A$CTR));
    END;
    IF CONTROL$FLAG THEN
        CALL WRITE$BYTE(CR);
```

```
    ELSE
        CALL WRITE$END$MARK;
END WRITE$FROM$MEMORY;

    /* * * * * * * * * RANDOM I-Ø PROCEDURES * * * * * * * * */

SET$RAN$POINTER: PROC;
 /* THIS PROCEDURE READS THE RANDOM KEY AND COMPUTES
    WHICH RECORD NEEDS TO BE AVAILABLE IN THE BUFFER
    THAT RECORD IS MADE AVAILABLE AND THE POINTERS
    SET FOR INPUT OR OUTPUT */
    DCL (BYTE$COUNT,TEMP,RECORD) ADDRESS,
         EXTENT BYTE;
    IF WRITE$MARK THEN CALL WRITE$RECORD;
    TEMP = CONVERT$TO$HEX(C$ADDR(3),C$BYTE(8));
    IF TEMP = Ø THEN
        DO;
            INVALID = TRUE;
            RETURN;
        END;
    BYTE$COUNT = (C$ADDR(2) + 2) * (TEMP - 1);
    RECORD = SHR(BYTE$COUNT,7);
    EXTENT = SHR(RECORD,7);
    IF EXTENT <> FCB$BYTE$A(EXTENT$OFFSET) THEN
        DO;
            CALL CLOSE(C$ADDR(Ø));
            FCB$BYTE$A(EXTENT$OFFSET) = EXTENT;
            IF OPEN(C$ADDR(Ø)) = 255 THEN
                DO;
                    IF SHR(CURRENT$FLAG,1) THEN
                        CALL MAKE(C$ADDR(Ø));
                    ELSE
                        DO;
                            INVALID = TRUE;
                            FCB$BYTE$A(EXTENT$OFFSET) = Ø;
                            IF OPEN(C$ADDR(Ø)) = 255 THEN
                                CALL FATAL$ERROR('OP');
                        END;
                END;
        END;
    BUFF$PTR = (BYTE$COUNT AND 7FH) + BUFF$START - 1;
    FCB$BYTE$A(32) = LOW(RECORD) AND 7FH;
    CALL READ$RECORD;
END SET$RAN$POINTER;

GET$REC$NUMBER: PROC ADDRESS;
    DCL (RECORD,LOGICAL$REC$NUM,BYTE$COUNT) ADDRESS;
    RECORD = FCB$BYTE$A(EXTENT$OFFSET);
    RECORD = SHL(RECORD,7) + FCB$BYTE$A(REC$NO);
    IF NOT SHR(CURRENT$FLAG,1) THEN RECORD = RECORD - 1;
    BYTE$COUNT = SHL(RECORD,7) + ((BUFF$PTR + 1)-BUFF$START);
```

```
    LOGICAL$REC$NUM = (BYTE$COUNT / (C$ADDR(2) + 2)) + 1;
    RETURN LOGICAL$REC$NUM;
END GET$REC$NUMBER;

SET$RELATIVE$KEY: PROC;
    DCL (REC$NUM, K) ADDRESS,
        (I,CNT) BYTE,
         J(4) ADDRESS DATA (10000,1000,100,10),
         BUFF(5) BYTE;
    REC$NUM = GET$REC$NUMBER;
    DO I = 0 TO 3;
    CNT = 0;
        DO WHILE REC$NUM >= (K := J(I));
            REC$NUM = REC$NUM - K;
            CNT = CNT + 1;
        END;
        BUFF(I)=CNT + '0';
    END;
    BUFF(4) = REC$NUM + '0';
    IF (I := C$BYTE(8)) <= 5 THEN
        CALL MOVE(.BUFF + 5 - I,RES(C$ADDR(3)),I);
    ELSE
        DO;
            CALL FILL(RES(C$ADDR(3)),I - 5,'0');
            CALL MOVE(.BUFF,RES(C$ADDR(3)) + I - 5,5);
        END;
END SET$RELATIVE$KEY;

WRT$EMPTY$REC: PROC;
    DO A$CTR = 1 TO C$ADDR(2);
        CALL WRITE$BYTE(HIGH$VALUE);
    END;
    CALL WRITE$END$MARK;
END WRT$EMPTY$REC;

WRITE$DUMMY$RECS: PROC(DIFFERENCE);
    DCL DIFFERENCE ADDRESS, COUNT BYTE;
    DO COUNT = 1 TO DIFFERENCE;
        CALL WRT$EMPTY$REC;
    END;
END WRITE$DUMMY$RECS;

BACK$ONE$EXTENT: PROC;
    CALL CLOSE(C$ADDR(0));
    IF (FCB$BYTE$A(EXTENT$OFFSET) :=
        FCB$BYTE$A(EXTENT$OFFSET)-1)=255 THEN
        CALL FATAL$ERROR('W?');
    IF OPEN(C$ADDR(0)) = 255 THEN
        DO;
            CALL FATAL$ERROR('OP');
            INVALID = TRUE;
```

```
            RETURN;
        END;
    FCB$BYTE$A(REC$NO) = 127;
END BACK$ONE$EXTENT;

BACK$ONE$RECORD: PROC;
    IF(BUFF$PTR := BUFF$PTR - (C$ADDR(2) + 2)) >=
        BUFF$START - 1 THEN
        DO;
            FCB$BYTE$A(REC$NO) = FCB$BYTE$A(REC$NO) - 1;
            RETURN;
        END;
    BUFF$PTR = BUFF$START - BUFF$PTR;
    DO WHILE BUFF$PTR > 129;
        BUFF$PTR = BUFF$PTR - 128;
        FCB$BYTE$A(REC$NO) = FCB$BYTE$A(REC$NO) - 1;
    END;
    BUFF$PTR = BUFF$END - BUFF$PTR;
    FCB$BYTE$A(REC$NO) = FCB$BYTE$A(REC$NO) - 2;
    IF FCB$BYTE$A(REC$NO) > 127 THEN
        DO;
           CALL BACK$ONE$EXTENT;
           IF INVALID THEN RETURN;
           CALL READ$RECORD;
           FCB$BYTE$A(REC$NO) = 127;
        END;
    ELSE
        DO;
           CALL READ$RECORD;
           FCB$BYTE$A(REC$NO) = FCB$BYTE$A(REC$NO) - 1;
        END;
END BACK$ONE$RECORD;

REWRITE$SEQ: PROC(FLAG);
     DCL FLAG BYTE;
     CALL BACK$ONE$RECORD;
     REWRITE$FLAG = TRUE;
     IF FLAG THEN CALL WRITE$FROM$MEMORY;
     ELSE CALL WRT$EMPTY$REC;  /* THIS IS A DELETE */
     CALL WRITE$RECORD;
     IF FCB$BYTE$A(REC$NO) = 0 THEN
         CALL BACK$ONE$EXTENT;
     ELSE
         FCB$BYTE$A(REC$NO) = FCB$BYTE$A(REC$NO) - 1;
     REWRITE$FLAG = FALSE;
     CALL READ$RECORD;
END REWRITE$SEQ;

CHECK$DIFFERENCE: PROC;
    DCL (DIFFERENCE,NEXT$RECORD,NEXT$KEY) ADDRESS;
    NEXT$RECORD = GET$REC$NUMBER;
```

```
    NEXT$KEY = CONVERT$TO$HEX(C$ADDR(3),C$BYTE(8));
    IF NEXT$RECORD > NEXT$KEY THEN CALL FATAL$ERROR('W2');
    DIFFERENCE = NEXT$KEY - NEXT$RECORD;
    IF DIFFERENCE > 0 THEN CALL WRITE$DUMMY$RECS(DIFFERENCE);
END CHECK$DIFFERENCE;

/* * * * * * * * * * * * * MOVES * * * * * * * * * * * * * * */

LOAD$INC: PROC;
    H$BYTE(CTR) = B$BYTE(CTR1);
    CTR1 = CTR1 + 1;
    CTR = CTR + 1;
END LOAD$INC;

CHECK$EDIT: PROC(CHAR);
    DCL CHAR BYTE;
    IF (CHAR = '0') OR (CHAR = '/') THEN CTR = CTR + 1;
    ELSE IF CHAR = 'B' THEN
        DO;
            H$BYTE(CTR) = ' ';
            CTR = CTR + 1;
        END;
    ELSE IF CHAR = 'A' THEN
        DO;
            IF NOT LETTER(B$BYTE(CTR1)) THEN
                CALL PRINT$ERROR('IC');
            CALL LOAD$INC;
        END;
    ELSE IF CHAR = '9' THEN
        DO;
            IF NOT NUMERIC (B$BYTE(CTR1)) THEN
                CALL PRINT$ERROR('IC');
            CALL LOAD$INC;
        END;
    ELSE CALL LOAD$INC;
END CHECK$EDIT;

/* * * * * * * * * * MACHINE ACTIONS * *  * * * * * * * * * */

STOP: PROC;
    CALL CRLF;
    DO CTR = 1 TO 4;
        CALL PRINT$CHAR(ERROR$CTR(CTR));
    END;
    CALL MON1(9,.(' EXECUTION ERRORS$'));
    CALL BOOTER;
END STOP;

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

    THE PROCEDURE BELOW CONTROLS THE EXECUTION OF THE CODE.
```

```
    IT DECODES EACH OP-CODE AND PERFORMS THE ACTIONS

* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

EXECUTE: PROC;
    DO FOREVER;
        DO CASE GET$OP$CODE;
              ;     /* CASE ZERO NOT USED */
 /* 01: ADD */
              CALL ADD;
 /* 02: SUB */
              DO;
                   SIGN0(0) = SIGN0(0) XOR 1;
                   CALL ADD;
              END;
 /* 03: MUL */
              DO;
                  DCL (I, X) BYTE;
                  CALL SET$MULT$DIV;
                  BASE = .R0;
                  CALL SHIFT$RIGHT(17);
                  BASE = .R1;
                  CALL SHIFT$RIGHT(1);
                  DEC$PT2 = DEC$PT0 + DEC$PT1;
                  I = 10;
                  DO INDEX = 1 TO 9;
                      CALL MULTIPLY(R1(I := I - 1) AND 0FH);
                      CALL MULTIPLY(SHR(R1(I),4));
                  END;
                  BASE = .R2;
                  CALL SHIFT$LEFT(17);
                  IF OVERFLOW THEN
                      IF (X := CTR + DEC$PT2) < 17 THEN
                          DEC$PT2 = 0;
                      ELSE
                          DO;
                              DEC$PT2 = X - 17;
                              OVERFLOW = FALSE;
                          END;
                  REG$LENGTH = 10;
                  CALL CHECK$RESULT;
              END;
 /* 04: DIV */
              CALL DIVIDE;
 /* 05: NEG */
              BRANCH$FLAG = NOT BRANCH$FLAG;
 /* 06: STP */
              CALL STOP;
 /* 07: STI */
              CALL STORE$IMMEDIATE;
 /* 08: EXT */
```

```
            IF RTN$BASE < HI$FREE$MEM THEN
                DO;
                    PROGRAM$COUNTER = RTN$PTR(0);
                    LOW$OFFSET = RTN$PTR(1);
                    HI$OFFSET = RTN$PTR(2);
                    RTN$BASE = RTN$BASE + 6;
                    CALL$TOP = CALL$BASE;
                    CALL$BASE = CALL$PTR(0);
                END;
/* 09: RND */
            DO;
                IF NOT OVERFLOW THEN
                    DO;
                        BASE = .R2;
                        IF (DEC$PT2 - C$BYTE(0)) > 0 THEN
                            DO;
                                CALL SHIFT$RIGHT(DEC$PT2 -
                                    C$BYTE(0));
                                DEC$PT2 = C$BYTE(0);
                            END;
                        ELSE
                            DO;
                                CALL SHIFT$LEFT(C$BYTE(0) -
                                    DECPT2);
                                DEC$PT2 = DEC$PT2 + CTR;
                            END;
                        CALL CHECK$RESULT;
                    END;
                CALL INC$PTR(1);
            END;
/* 10: RET */
            DO;
                IF C$ADDR(0) <> 0 THEN
                    DO;
                        A$CTR = C$ADDR(0);
                        C$ADDR(0) = 0;
                        PROGRAM$COUNTER = A$CTR;
                    END;
                ELSE CALL INC$PTR(2);
            END;
/* 11: CLS */
            DO;
                CALL SET$I$O;
                IF WRITE$MARK THEN
                    DO;
                        IF NOT SHR(CURRENT$FLAG,2) THEN
                            CALL WRITE$BYTE(TERMINATOR);
                        CALL WRITE$RECORD;
                    END;
                ELSE CALL SET$DMA;
                CALL CLOSE(C$ADDR(0));
```

```
                CURRENT$FLAG,FCB$BYTE$A(FLAG$OFFSET) = 0;
                CALL INC$PTR(2);
            END;
/* 12: SER */
            IF OVERFLOW THEN
                DO;
                    CALL INC$PTR(3);
                    OVERFLOW = FALSE;
                END;
/* 13: BRN */
            PROGRAM$COUNTER = C$ADDR(0);
/* 14: OPN */
            DO;
                CALL OPEN$FILE(1);
                CALL READ$RECORD;
            END;
/* 15: OP1 */
            CALL OPEN$FILE(2);
/* 16: OP2 */
            DO;
                CALL OPEN$FILE(4);
                CALL READ$RECORD;
            END;
/* 17: RGT */
            DO;
                IF NOT SIGN0(2) THEN
                    BRANCH$FLAG = NOT BRANCH$FLAG;
                CALL COND$BRANCH(0);
            END;
/* 18: RLT */
            DO;
                IF SIGN0(2) AND NOT R2$ZERO THEN
                    BRANCH$FLAG = NOT BRANCH$FLAG;
                CALL COND$BRANCH(0);
            END;
/* 19: REQ */
            DO;
                IF R2$ZERO THEN
                    BRANCH$FLAG = NOT BRANCH$FLAG;
                CALL COND$BRANCH(0);
            END;
/* 20: INV */
            CALL INCR$OR$BRANCH(INVALID);
/* 21: EOR */
            CALL
                INCR$OR$BRANCH(FCB$BYTE$A(EOF$FLAG$OFFSET));
/* 22: PAG */
            DO;
                DCL I BYTE;
                CALL SET$I$0;
                IF C$BYTE(2) < 100 THEN
```

```
                    DO I = 1 TO C$BYTE(2);
                        CALL WRITE$BYTE(LF);
                    END;
                ELSE
                    CALL WRITE$BYTE(TOP$OF$PAGE);
                IF C$BYTE(3) = WTF THEN
                    CONTROL$FLAG = TRUE;
                CALL INC$PTR(3);
            END;
/* 23: ACC */
            CALL ACCEPT;
/* 24: STD */
            DO;
                TEMP = C$BYTE(3);
                C$BYTE(3) = 0;
                CALL DISPLAY;
                CALL PRINT(.(LF,'OPERATOR ENTER A <CR> TO
                                CONTINUE$'));
                CALL PRINT(.(TAB,' OR ENTER AN "S" TO
                                  TERMINATE.$'));
                CHAR = 0;
                DO WHILE (CHAR <> CR) AND (CHAR <> 'S');
                    CALL PRINT(.(CR,LF,'?$'));
                    CHAR = MON2(1,0);
                END;
                IF CHAR = CR THEN
                   DO;
                      PROGRAM$COUNTER = PROGRAM$COUNTER - 1;
                      C$BYTE(0) = TEMP;
                   END;
                ELSE CALL STOP;
            END;
/* 25: LDI */
            DO;
              C$ADDR(2) =
               CONVERT$TO$HEX(RES(C$ADDR(0)),C$BYTE(2)) + 1;
              CALL INC$PTR(3);
            END;
/* 26: DIS */
            CALL DISPLAY;
/* 27: DEC */
            DO;
                IF C$ADDR(0) <> 0 THEN
                    C$ADDR(0) = C$ADDR(0) - 1;
                IF C$ADDR(0) = 0 THEN
                    PROGRAM$COUNTER = C$ADDR(1);
                ELSE CALL INC$PTR(4);
            END;
/* 28: STO */
            DO;
                CALL STORE$NUMERIC;
```

```
                CALL INC$PTR(4);
            END;
/* 29: ST1 */
            DO;
                CALL STORE$NUMERIC;
                CALL SET$ZONE(RES(C$ADDR(0)));
            END;
/* 30: ST2 */
            DO;
              CALL STORE$NUMERIC;
              CALL SET$ZONE(RES(C$ADDR(0)) + C$BYTE(2) - 1);
            END;
/* 31: ST3 */
            DO;
                CALL CHECK$DECIMAL;
                BASE = RES(C$ADDR(0)) + C$BYTE(2) - 1;
                CALL STORE$AS$CHAR(C$BYTE(2) - 1);
                CALL SET$SIGN$SEP(RES(C$ADDR(0)));
            END;
/* 32: ST4 */
            DO;
                CALL CHECK$DECIMAL;
                BASE = RES(C$ADDR(0)) + C$BYTE(2) - 2;
                CALL STORE$AS$CHAR(C$BYTE(2) - 1);
                CALL SIT$SIGN$SEP
                    (RES(C$ADDR(0)) + C$BYTE(2) - 1);
            END;
/* 33: ST5 */
            DO;
                CALL CHECK$DECIMAL;
                IF SIGN0(2) = 0 THEN
                    R2(9) = R2(9) OR 01H;
                IF C$BYTE(4) <> SER OR NOT OVERFLOW TEEN
                    DO;
                        CTR = C$BYTE(2) / 2 + 1;
                        CALL MOVE
                        (.R2 + 10 - CTR,RES(C$ADDR(0)),CTR);
                    END;
                CALL INC$PTR(4);
            END;
/* 34: LOD */
            CALL LOAD$NUM$LIT;
/* 35: LD1 */
            CALL LOAD$NUMERIC;
/* 36: LD2 */
            DO;
                HOLD = RES(CADDR(0));
                IF CHECK$FOR$SIGN(H$BYTE(0)) THEN
                  DO;
                    CALL SET$LOAD(POSITIVE);
                    CALL LOAD$NUMBERS(C$ADDR(0),C$BYTE(2));
```

```
                END;
            ELSE
                DO;
                    TEMP = H$BYTE(0);
                    CALL SET$LOAD(NEGITIVE);
                    CALL LOAD$NUMBERS
                        (C$ADDR(0) + 1,C$BYTE(2) - 1);
                    CALL LOAD$A$CHAR(TEMP - ZONE);
                END;
        END;
/* 37: LD3 */
        DO;
            DCL I BYTE;
            HOLD = RES(C$ADDR(0));
            IF CHECK$FOR$SIGN(
                CTR := H$BYTE(I := C$BYTE(2) - 1)) THEN
                DO;
                    CALL SET$LOAD(POSITIVE);
                    I = I + 1;
                END;
            ELSE
                DO;
                    CALL SET$LOAD(NEGITIVE);
                    CALL LOAD$A$CHAR(CTR - ZONE);
                END;
            CALL LOAD$NUMBERS(C$ADDR(0),I);
        END;
/* 38: LD4 */
        DO;
            HOLD = RES(C$ADDR(0));
            IF (H$BYTE(0) = '+') THEN
                CALL SET$LOAD(POSITIVE);
            ELSE CALL SET$LOAD(NEGITIVE);
            CALL LOAD$NUMBERS(C$ADDR(0) + 1,
                              C$BYTE(2) - 1);
        END;
/* 39: LD5 */
        DO;
            HOLD = RES(C$ADDR(0));
            IF H$BYTE(C$BYTE(2) - 1) = '+' THEN
                CALL SET$LOAD(POSITIVE);
            ELSE CALL SET$LOAD(NEGITIVE);
            CALL LOAD$NUMBERS(C$ADDR(0),C$BYTE(2) - 1);
        END;
/* 40: LD6 */
        DO;
            DCL I BYTE;
            HOLD = RES(C$ADDR(0));
            IF H$BYTE (I := C$BYTE(2) / 2) THEN
                CALL SET$LOAD(NEGITIVE);
            ELSE CALL SET$LOAD(POSITIVE);
```

```
                BASE = BASE + 9 - I;
                DO CTR = 0 TO I;
                    B$BYTE(CTR) = H$BYTE(CTR);
                END;
                B$BYTE(I) = B$BYTE(I) AND 0F0H;
                CALL INC$PTR(5);
            END;
/* 41: PER */
            DO;
                BASE = C$ADDR(1) + 1;
                B$ADDR(0) = C$ADDR(2);
                PROGRAM$COUNTER = C$ADDR(0);
            END;
/* 42: CNU */
            CALL COMP$NUM$UNSIGNED;
/* 43: CNS */
            CALL COMP$NUM$SIGN;
/* 44: CAL */
            CALL COMP$ALPHA;
/* 45: RWS */
            DO;
                CALL SET$I$0;
                IF NOT SHR(CURRENT$FLAG,2) THEN
                    CALL FATAL$ERROR('W6');
                IF NOT FCB$BYTE$A(EOF$FLAG$OFFSET) THEN
                    CALL REWRITE$SEQ(1);
                CALL INC$PTR(6);
            END;
/* 46: DLS */
            DO;
                CALL SET$I$0;
                IF NOT SHR(CURRENT$FLAG,2) THEN
                    CALL FATAL$ERROR('W6');
                IF NOT FCB$BYTE$A(EOF$FLAG$OFFSET) THEN
                    CALL REWRITE$SEQ(0);
                CALL INC$PTR(6);
            END;
/* 47: RDF */
            DO;
                CALL SET$I$0;
                IF NOT CURRENT$FLAG THEN
                    CALL FATAL$ERROR('W5');
                IF NOT FCB$BYTE$A(EOF$FLAG$OFFSET) THEN
                    CALL READ$TO$MEMORY;
                CALL INC$PTR(6);
            END;
/* 48: WTF */
            DO;
                IF C$BYTE(6) = PAG THEN
                    CONTROL$FLAG = TRUE;
                CALL SET$I$0;
```

```
                IF NOT SHR(CURRENT$FLAG,1) THEN
                    CALL FATAL$ERROR('W3');
                CALL WRITE$FROM$MEMORY;
                CALL INC$PTR(6);
                CONTROL$FLAG = FALSE;
            END;
/* 49: RVL */
            DO;
                CALL READ$VARIABLE;
                CALL INC$PTR(6);
            END;
/* 50: WVL */
            DO;
                CALL WRITE$VARIABLE;
                CALL INC$PTR(6);
            END;
/* 51: SCR */
            DO;
              SUBSCRIPT(C$BYTE(7)) = C$ADDR(0) + C$ADDR(1) *
                  (CONVERT$TO$HEX(C$ADDR(2),C$BYTE(6)) - 1);
              CALL INC$PTR(8);
            END;
/* 52: SGT */
            CALL STRING$COMPARE(1);
/* 53: SLT */
            CALL STRING$COMPARE(0);
/* 54: SEQ */
            CALL STRING$COMPARE(2);
/* 55: MOV */
            DO;
              CALL MOVE(RES(C$ADDR(1)),RES(C$ADDR(0)),
                                          C$ADDR(2));
              IF C$ADDR(3) <> 0 THEN
                DO;
                 CALL FILL(RES(C$ADDR(0)) + C$ADDR(2),
                              C$ADDR(3),FILLER);
                END;
              CALL INC$PTR(8);
            END;
/* 56: RRS */
            DO;
                DCL H$FLAG BYTE;
                H$FLAG = TRUE;
                CALL SET$I$0;
                IF SHR(CURRENT$FLAG,1) THEN
                    CALL FATAL$ERROR('W5');
                DO WHILE (NOT FCB$BYTE$A(EOF$FLAG$OFFSET))
                    AND H$FLAG;
                    H$FLAG = FALSE;
                    CALL SET$RELATIVE$KEY;
                    CALL READ$TO$MEMORY;
```

```
                    IF B$BYTE(0) = HIGH$VALUE THEN
                        H$FLAG = TRUE;
                END;
                CALL INC$PTR(9);
            END;
/* 57: WRS */
            DO;
                CALL SET$I$0;
                IF NOT SHR(CURRENT$FLAG,1) THEN
                    CALL FATAL$ERROR('W1');
                CALL CHECK$DIFFERENCE;
                CALL SET$RELATIVE$KEY;
                CALL WRITE$FROM$MEMORY;
                CALL INC$PTR(9);
            END;
/* 58: RRR */
            DO;
                CALL SET$I$0;
                IF SHR(CURRENT$FLAG,1) THEN
                    CALL FATAL$ERROR('W5');
                CALL SET$RAN$POINTER;
                IF NOT INVALID THEN
                    CALL READ$TO$MEMORY;
                IF INVALID THEN
                    FCB$BYTE$A(EOF$FLAG$OFFSET) = FALSE;
                CALL INC$PTR(9);
            END;
/* 59: WRR */
            DO;
                DCL DIFFERENCE ADDRESS;
                CALL SET$I$0;
                IF SHR(CURRENT$FLAG,1) THEN
                    DO;
                        CALL CHECK$DIFFERENCE;
                        CALL SET$RELATIVE$KEY;
                        CALL WRITE$FROM$MEMORY;
                    END;
                ELSE
                  DO;
                   IF SHR(CURRENT$FLAG,2) THEN
                    DO;
                      CALL SET$RAN$POINTER;
                      IF NOT INVALID THEN
                        IF (BUFF$BYTE(1)) = HIGH$VALUE THEN
                          DO;
                            REWRITE$FLAG = TRUE;
                            FCB$BYTE$A(REC$NO) =
                                FCB$BYTE$A(REC$NO) - 1;
                            CALL WRITE$FROM$MEMORY;
                            REWRITE$FLAG = FALSE;
                          END;
```

```
                        ELSE CALL FATAL$ERROR('W4');
                      ELSE CALL FATAL$ERROR('W3');
                    END;
                  END;
                CALL INC$PTR(9);
              END;
/* 60: RWR */
            DO;
              CALL SET$I$0;
              IF NOT SHR(CURRENT$FLAG,2) THEN
                  CALL FATAL$ERROR('W6');
              REWRITE$FLAG = TRUE;
              CALL BACK$ONE$RECORD;
              IF NOT INVALID THEN CALL WRITE$FROM$MEMORY;
              REWRITE$FLAG = FALSE;
              CALL INC$PTR(9);
            END;
/* 61: DLR */
            DO;
                CALL SET$I$0;
                IF NOT SHR(CURRENT$FLAG,2) THEN
                    CALL FATAL$ERROR('W6');
                CALL SET$RAN$POINTER;
                REWRITE$FLAG = TRUE;
                IF NOT INVALID THEN
                    DO;
                        FCB$BYTE$A(REC$NO) =
                            FCB$BYTE$A(REC$NO) - 1;
                        CALL WRT$EMPTY$REC;
                    END;
                REWRITE$FLAG = FALSE;
                CALL INC$PTR(9);
            END;
/* 62: MED */
            DO;
                HOLD = RES(C$ADDR(0));
                CALL MOVE(RES(C$ADDR(3)),HOLD,C$ADDR(4));
                BASE = RES(C$ADDR(1));
                CTR,CTR1 = 0;
                DO WHILE (CTR1 < C$ADDR(2))
                     AND (CTR  < C$ADDR(4));
                    CALL CHECK$EDIT(H$BYTE(CTR));
                END;
                DO WHILE CTR < C$ADDR(4);
                    IF H$BYTE(CTR) = 'X' OR
                       H$BYTE(CTR) = 'A' OR
                       H$BYTE(CTR) = '9' THEN
                           H$BYTE(CTR) = FILLER;
                    ELSE IF H$BYTE(CTR) = 'B' THEN
                              H$BYTE(CTR) = ' ';
                    CTR = CTR + 1;
```

```
                END;
                CALL INC$PTR(10);
            END;
 /* 63: MNE */
            CALL MOVE$NUM$EDITED;
 /* 64: SBR */
            DO;
                RTN$BASE = RTN$BASE - 6;
                RTN$PTR(0) = PROGRAM$COUNTER + 6;
                RTN$PTR(1) = LOW$OFFSET;
                RTN$PTR(2) = HI$OFFSET;
                LOW$OFFSET = C$ADDR(1);
                HI$OFFSET = C$ADDR(2);
                PROGRAM$COUNTER = C$ADDR(0);
            END;
 /* 65: GDP */
            DO;
                DCL OFFSET BYTE;
                OFFSET = CONVERT$TO$HEX(RES(C$ADDR(1)),
                                                C$BYTE(1));
                IF OFFSET > C$BYTE(0) OR OFFSET < 1 THEN
                    DO;
                        CALL PRINT$ERROR('GD');
                        CALL INC$PTR(SHL(C$BYTE(0),1) + 4);
                    END;
                ELSE PROGRAM$COUNTER = C$ADDR(OFFSET + 1);
            END;
 /* 66: PAR */
            DO;
                HOLD = CALL$TOP;
                CALL$TOP = CALL$TOP + SHL(C$ADDR(0),1) + 2;
                IF CALL$TOP > RTN$BASE - 7 THEN
                    CALL FATAL$ERROR('CO');
                H$ADDR(0) = CALL$BASE;
                DO CTR = 1 TO C$ADDR(0);
                    H$ADDR(CTR) = RES(C$ADDR(CTR));
                END;
                CALL$BASE = HOLD;
                CALL INC$PTR(SHL(C$ADDR(0),1) + 2);
            END;
        END; /* END OF CASE STATEMENT */
    END; /* END OF DO FOREVER */
END EXECUTE;

/* * * * * * * PROGRAM EXECUTION STARTS HERE * * * * * * * */

CALL MOVE(00FCH,.HI$FREE$MEM,4);
HI$FREE$MEM = MAX$MEMORY - HI$FREE$MEM;
LOW$FREE$MEM = CODE$START + LOW$FREE$MEM + 2;
RTN$BASE = HI$FREE$MEM;
CALL$TOP,CALL$BASE = LOW$FREE$MEM;
```

```
CALL PRINT(.('NPS MICRO-COBOL INTERPRETER VERSION 2.0$'));
CALL PRINT(.('EXECUTION BEGINS$'));
BASE = CODE$START;
PROGRAM$COUNTER = B$ADDR(0);
CALL EXECUTE;
END;
```

COMPUTER LISTING FOR MODULE READER NPS MICRO-COBOL

```
$ TITLE('NPS MICRO-COBOL COMPILER READER') PAGEWIDTH(80)
     PAGELENGTH(60)
READER: DO;

             /*     COBOL COMPILER - READER     */

            /*     NORMALLY LOCATED AT B000H     */

         /*     GLOBAL DECLARATIONS AND LITERALS     */

    /* THIS PROGRAM IS LOADED IN WITH THE PART 1 PROGRAM
    AND IS CALLED WHEN PART 1 IS FINISHED.  THIS PROGRAM
    OPENS THE PART2.COM FILE THAT CONTAINS THE CODE FOR
    PART 2 OF THE COMPILER, AND READS IT INTO CORE. AT
    THE END OF THE READ OPERATION, CONTROL IS PASSED TO
    THE SECOND PART OF THE PROGRAM.                 */

DECLARE
    LIT      LITERALLY     'LITERALLY',
    ADDR     ADDRESS       INITIAL(100H),
    DCL      LIT           'DECLARE',
    FCB(33) BYTE           INITIAL(0,'PART2   COM',
         0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0),
    I        ADDRESS,
    PROC     LIT           'PROCEDURE',
    START    LIT           '100H';

MON1: PROC(F,A) EXTERNAL;
    DCL F BYTE, A ADDRESS;
END MON1;

MON2: PROC(F,A) BYTE EXTERNAL;
    DCL F BYTE, A ADDRESS;
END MON2;

BOOT: PROC EXTERNAL;
END BOOT;

OPEN: PROC(FCB) BYTE;
    DCL FCB ADDRESS;
    RETURN MON2(15,FCB);
END OPEN;

READ: PROC(ADDR) BYTE;
    DCL ADDR ADDRESS;
    CALL MON1 (26,ADDR);     /* SET DMA ADDRESS */
    RETURN MON2 (20,.FCB); /* READ, AND RETURN ERROR CODE */
END READ;
```

```
ERROR: PROC(CODE);
    DCL CODE ADDRESS;
    CALL MON1(2,(HIGH(CODE)));
    CALL MON1(2,(LOW(CODE)));
    CALL BOOT;
END ERROR;

    /*    PROGRAM EXECUTION STARTS HERE    */

    CALL MON1 (26,0100H);
    IF OPEN(.FCB) = 255 THEN CALL ERROR('02');
    I = 0100H;
    DO WHILE READ(I) = 0;
        I = I + 0080H;
    END;
    CALL MON1 (26, 0080H);       /* RESET DMA ADDRESS */
    CALL ADDR;
END;
```

```
$ TITLE('NPS MICRO-COBOL COMPILER BUILD') PAGEWIDTH(80)
    PAGELENGTH(60)
BUILD: DO;

                /*    COBOL COMPILER - BUILD    */

              /*    NORMALLY LOCATED AT 103H    */

          /*    GLOBAL DECLARATIONS AND LITERALS    */

    /* THIS PROGRAM TAKES THE CODE OUTPUT FROM THE COBOL
    COMPILER AND BUILDS THE ENVIRONMENT FOR THE COBOL
    INTERPRETER */


DECLARE
    LIT                 LITERALLY       'LITERALLY',
    TRUE                LIT             '1',
    ADDR                ADDRESS         INITIAL(100H),
    BASE                ADDRESS,
    B$ADDR              BASED           BASE ADDRESS,
    B$BYTE              BASED           BASE (4) BYTE,
    BOOT                LIT             '0',
    BUFF$END            LIT             '100H',
    CHAR                BASED           ADDR BYTE,
    CODE$CTR            ADDRESS,
    C$ADDR              BASED           CODE$CTR ADDRESS,
    C$BYTE              BASED           CODE$CTR BYTE,
    CODE$NOT$SET        BYTE            INITIAL(TRUE),
    CUR$SYM             ADDRESS,
    DCL                 LIT             'DECLARE',
    EXT                 LIT             '08H',
    FALSE               LIT             '0',
    FCB                 ADDRESS         INITIAL(5CH),
    FCB$BYTE            BASED FCB       BYTE,
    FCB$BYTE$A          BASED FCB (33)  BYTE,
    FILE$TYPE (*)       BYTE            DATA('.CIN$'),
    FOREVER             LIT             'WHILE TRUE',
    FREE$STORAGE        ADDRESS,
    HASH$MASK           BYTE            INITIAL(0FH),
    I                   BYTE,
    INTERP$ADDRESS      ADDRESS         INITIAL(3500H),
    INTERP$CONTENT      BASED           INTERP$ADDRESS ADDRESS,
    INTERP$FCB(33)      BYTE            INITIAL(0,'CINTERP COM',
                                                0,0,0,0),
    I$BYTE              BASED           INTERP$ADDRESS (2) BYTE,
    HI$OFFSET           ADDRESS         INITIAL(00H),
```

```
    LOW$OFFSET      ADDRESS         INITIAL(00H),
    LOADED          LIT             '10H',
    MAX$MEMORY      ADDRESS         INITIAL(1C80H),
    NEXT$SYM        ADDRESS,
    NEXT$SYM$ENTRY  BASED NEXT$SYM ADDRESS,
    POINT           ADDRESS,
    COLLISION       BASED POINT     ADDRESS,
    PROC            LIT             'PROCEDURE',
    PROC$NAME(8)    BYTE,
    READER$LOCATION ADDRESS         INITIAL(1C80H),
    STP             LIT             '06H',
    SUB$FLAG        BYTE            INITIAL(FALSE),
    SYMBOL          BASED CUR$SYM (1) BYTE,
    SYMBOL$ADDR     BASED CUR$SYM (1) ADDRESS,
    TOP$OF$MEMORY   ADDRESS         INITIAL(0B100H);

MON1: PROC(F,A) EXTERNAL;
    DCL F BYTE, A ADDRESS;
END MON1;

MON2: PROC(F,A) BYTE EXTERNAL;
    DCL F BYTE, A ADDRESS;
END MON2;

PRINT$CHAR: PROC(CHAR);
    DCL CHAR BYTE;
    CALL MON1(2,CHAR);
END PRINT$CHAR;

CRLF: PROC;
    CALL PRINT$CHAR(13);
    CALL PRINT$CHAR(10);
END CRLF;

PRINT: PROC(A);
    DCL A ADDRESS;
    CALL MON1(9,A);
END PRINT;

PRINT$NAME: PROC(ADDR);
    DCL ADDR ADDRESS;
    BASE = ADDR;
    I = 255;
    CALL CRLF;
    DO WHILE(B$BYTE(I := I + 1) <> ' ') AND (I < 8);
        CALL PRINT$CHAR(B$BYTE(I));
    END;
END PRINT$NAME;

OPEN: PROC(A) BYTE;
    DCL A ADDRESS;
```

```
    RETURN MON2(15,A);
END OPEN;

CLOSE: PROC(FCB);
    DCL FCB ADDRESS;
    IF MON2(16,FCB) = 255 THEN
        DO;
            CALL CRLF;
            CALL PRINT(.('CLOSE ERROR ON MODULE $'));
            CALL PRINT$NAME(FCB + 1);
        END;
END CLOSE;

REBOOT: PROC;
    ADDR = BOOT;
    CALL ADDR;
END REBOOT;

FATAL$ERROR: PROC(REASON);
    DCL REASON ADDRESS;
    CALL CRLF;
    CALL PRINT$CHAR(HIGH(REASON));
    CALL PRINT$CHAR(LOW (REASON));
    CALL PRINT$NAME(FCB + 1);
    CALL PRINT(.FILE$TYPE);
    CALL REBOOT;
END FATAL$ERROR;

MOVE: PROC(FROM, DEST, COUNT);
    DCL (FROM,DEST,COUNT) ADDRESS,
         (F BASED FROM,D BASED DEST) BYTE;
    DO WHILE(COUNT := COUNT - 1) <> 0FFFFH;
         D = F;
         FROM = FROM + 1;
         DEST = DEST + 1;
    END;
END MOVE;

FILL: PROC(ADDR,CHAR,COUNT);
    DCL ADDR ADDRESS,
        (CHAR,COUNT,DEST BASED ADDR) BYTE;
    DO WHILE (COUNT := COUNT - 1) <> 0FFH;
        DEST = CHAR;
        ADDR = ADDR + 1;
    END;
END FILL;

GET$CHAR: PROC BYTE;
    IF (ADDR := ADDR + 1) >= BUFF$END THEN
        DO;
            IF MON2(20,FCB) <> 0 THEN
```

```
                DO;
                    CALL CRLF;
                    CALL PRINT(.('END OF INPUT$'));
                    CALL REBOOT;
                END;
            ADDR = 80H;
        END;
    RETURN CHAR;
END GET$CHAR;
NEXT$CHAR: PROC;
    CHAR = GET$CHAR;
END NEXT$CHAR;

STORE: PROC(COUNT);
    DCL COUNT BYTE;
    IF CODE$NOT$SET THEN
        DO;
            CALL CRLF;
            CALL PRINT(.('CODE ERROR$'));
            CALL NEXT$CHAR;
            RETURN;
        END;
    DO I = 1 TO COUNT;
        C$BYTE = CHAR;
        CALL NEXT$CHAR;
        CODE$CTR = CODE$CTR + 1;
    END;
END STORE;

INIT$LOAD$TABLE: PROC;
    FREE$STORAGE  = .MEMORY;
    CALL FILL(FREE$STORAGE,0,34);
    NEXT$SYM = FREE$STORAGE + 32;
    NEXT$SYM$ENTRY = 0;
END INIT$LOAD$TABLE;

BUILD$SYMBOL: PROC;
    DCL TEMP ADDRESS;
    TEMP = NEXT$SYM;
    IF (NEXT$SYM := .SYMBOL(17)) > MAX$MEMORY THEN
        CALL FATAL$ERROR('PS');
    CALL FILL(TEMP,0,17);
END BUILD$SYMBOL;

MATCH: PROC;
    DCL (HOLD,I) BYTE;
    HOLD = 0;
    DO I = 0 TO 7;
        HOLD = HOLD + PROC$NAME(I);
    END;
    POINT = FREE$STORAGE + SHL((HOLD AND HASH$MASK),1);
```

```
    DO FOREVER;
        IF COLLISION = 0 THEN
            DO;
                CUR$SYM,COLLISION = NEXT$SYM;
                CALL BUILD$SYMBOL;
                DO I = 0 TO 7;
                    SYMBOL(I + 8) = PROC$NAME(I);
                END;
                RETURN;
            END;
        ELSE
            DO;
                CUR$SYM = COLLISION;
                I = 0;
                DO WHILE SYMBOL(I + 8) = PROC$NAME(I);
                    IF (I := I + 1) > 7 THEN
                        DO;
                            CUR$SYM = COLLISION;
                            RETURN;
                        END;
                END;
            END;
        POINT = COLLISION;
    END;
END MATCH;

STUFF: PROC;
    DCL (HOLD,TEMP) ADDRESS;
    HOLD = SYMBOL$ADDR(1);
    BASE = .TEMP;
    B$BYTE(0) = GET$CHAR;
    B$BYTE(1) = GET$CHAR;
    SYMBOL$ADDR(1) = CODE$CTR + TEMP - INTERP$ADDRESS;
    DO WHILE HOLD <> 0;
        BASE = HOLD;
        HOLD = B$ADDR;
        DO I = 1 TO 3;
            B$ADDR = SYMBOL$ADDR(I);
            BASE = BASE + 2;
        END;
    END;
    CODE$CTR = SYMBOL$ADDR(1);
END STUFF;

COMPUTE$OFFSETS: PROC;
    DCL TEMP ADDRESS;
    BASE = .TEMP;
    B$BYTE(0) = GET$CHAR;
    B$BYTE(1) = GET$CHAR;
    HI$OFFSET = HI$OFFSET + (TOP$OF$MEMORY - TEMP + 1);
    LOW$OFFSET = CODE$CTR - INTERP$ADDRESS - 2;
```

```
END COMPUTE$OFFSETS;

SUBR: PROC;
    DCL I BYTE;
    CALL STORE(1);
    DO I = 0 TO 7;
        PROC$NAME(I) = CHAR;
        CALL NEXT$CHAR;
    END;
    CALL MATCH;
    DO I = 1 TO 3;
        C$ADDR = SYMBOL$ADDR(I);
        CODE$CTR = CODE$CTR + 2;
    END;
    IF SYMBOL(LOADED) = 0 THEN
        SYMBOL$ADDR(1) = CODE$CTR - 6;
END SUBR;

GO$DEPENDING: PROC;
    CALL STORE(1);
    CALL STORE(SHL(CHAR,1) + 4);
END GO$DEPENDING;

PARAMETERS: PROC;
    CALL STORE(1);
    CALL STORE(SHL(CHAR,1) + 2);
END PARAMETERS;

BACK$STUFF: PROC;
    DCL (HOLD,STUFF) ADDRESS;
    BASE = .HOLD;
    DO I = 0 TO 3;
        B$BYTE(I) = GET$CHAR;
    END;
    DO FOREVER;
        BASE = HOLD + LOW$OFFSET;
        HOLD = B$ADDR;
        B$ADDR = STUFF;
        IF HOLD = 0 THEN
            DO;
                CALL NEXT$CHAR;
                RETURN;
            END;
    END;
END BACK$STUFF;

INITIALIZE: PROC;
    DCL (COUNT,WHERE,HOW$MANY) ADDRESS;
    BASE = .WHERE;
    DO I = 0 TO 3;
        B$BYTE(I) = GET$CHAR;
```

```
    END;
    IF WHERE > TOP$OF$MEMORY - HI$OFFSET THEN
        BASE = WHERE - HI$OFFSET - 1;
    ELSE
        BASE = WHERE + LOW$OFFSET - 1;
    DO COUNT = 1 TO HOW$MANY;
        B$BYTE(COUNT) = GET$CHAR;
    END;
    CALL NEXT$CHAR;
END INITIALIZE;

TERMINATE: PROC;
    DCL I BYTE, TEMP ADDRESS;
    IF SUB$FLAG THEN C$BYTE = EXT;
    ELSE C$BYTE = STP;
    CODE$CTR = CODE$CTR + 1;
    I = 0FFH;
    CALL PRINT$NAME(FCB + 1);
    CALL PRINT(.(' LOADED$'));
    SUB$FLAG = FALSE;
    DO I = 0 TO 15;
        POINT = FREESTORAGE + 2 * I;
        DO WHILE COLLISION <> 0;
            CUR$SYM = COLLISION;
            IF SYMBOL(LOADED) = 0 THEN
                DO;
                    CODE$NOT$SET,SYMBOL(LOADED),SUB$FLAG =
                        TRUE;
                    CALL COMPUTE$OFFSETS;
                    SYMBOL$ADDR(2) = LOW$OFFSET;
                    SYMBOL$ADDR(3) = HI$OFFSET;
                    CALL CLOSE(FCB);
                    CALL MOVE(.SYMBOL(8),FCB + 1,8);
                    FCB$BYTE$A(32) = 0;
                    CALL FILL(FCB + 12,0,4);
                    ADDR = 100H;
                    IF OPEN(FCB) = 255 THEN
                        CALL FATAL$ERROR('OP');
                    CALL NEXT$CHAR;
                    RETURN;
                END;
            POINT = COLLISION;
        END; /* DO WHILE COLLISION <> 0 */
    END; /* DO I = 0 TO 15 */
END TERMINATE;

START$CODE: PROC;
    CODE$NOT$SET = FALSE;
    IF SUB$FLAG THEN CALL STUFF;
    ELSE
        DO;
```

```
            I$BYTE(0) = GET$CHAR;
            I$BYTE(1) = GET$CHAR;
            CODE$CTR = INTERP$CONTENT;
        END;
    CALL NEXT$CHAR;
END START$CODE;

BUILD: PROC;
    DCL
    F2   LIT  '9',
    F3   LIT  '10',
    F4   LIT  '22',
    F5   LIT  '26',
    F6   LIT  '34',
    F7   LIT  '41',
    F8   LIT  '51',
    F9   LIT  '51',
    F10  LIT  '56',
    F11  LIT  '62',
    F12  LIT  '63',
    F13  LIT  '63',
    SBR  LIT  '64',
    GDP  LIT  '65',
    PAR  LIT  '66',
    INT  LIT  '67',
    BST  LIT  '68',
    TER  LIT  '69',
    SCD  LIT  '70';

    DO FOREVER;
            IF CHAR <  F2 THEN CALL STORE(1);
        ELSE IF CHAR <  F3 THEN CALL STORE(2);
        ELSE IF CHAR <  F4 THEN CALL STORE(3);
        ELSE IF CHAR <  F5 THEN CALL STORE(4);
        ELSE IF CHAR <  F6 THEN CALL STORE(5);
        ELSE IF CHAR <  F7 THEN CALL STORE(6);
        ELSE IF CHAR <  F8 THEN CALL STORE(7);
        ELSE IF CHAR <  F9 THEN CALL STORE(8);
        ELSE IF CHAR < F10 THEN CALL STORE(9);
        ELSE IF CHAR < F11 THEN CALL STORE(10);
        ELSE IF CHAR < F12 THEN CALL STORE(11);
        ELSE IF CHAR < F13 THEN CALL STORE(12);
        ELSE IF CHAR < SBR THEN CALL STORE(13);
        ELSE IF CHAR = SBR THEN CALL SUBR;
        ELSE IF CHAR = GDP THEN CALL GO$DEPENDING;
        ELSE IF CHAR = PAR THEN CALL PARAMETERS;
        ELSE IF CHAR = BST THEN CALL BACK$STUFF;
        ELSE IF CHAR = INT THEN CALL INITIALIZE;
        ELSE IF CHAR = TER THEN
            DO;
                CALL TERMINATE;
```

```
                IF NOT SUB$FLAG THEN
                    DO;
                        CALL COMPUTE$OFFSETS;
                        CALL CLOSE(FCB);
                        RETURN;
                    END;
            END;
        ELSE IF CHAR = SCD THEN CALL START$CODE;
        ELSE
            DO;
                CALL CRLF;
                CALL PRINT(.('LOAD ERROR$'));
                CALL NEXT$CHAR;
            END;
    END;
END BUILD;

    /* PROGRAM EXECUTION STARTS HERE */

CALL CRLF;
CALL PRINT(.('NPS MICRO-COBOL LOADER VERS 2.0$'));
FCB$BYTE$A(32) = 0;
CALL MOVE(.('CIN',0,0,0,0),FCB + 9,7);
IF OPEN(FCB) = 255 THEN
    DO;
        CALL CRLF;
        CALL PRINT$NAME(FCB + 1);
        CALL PRINT(.FILE$TYPE);
        CALL REBOOT;
    END;
CALL NEXT$CHAR;
CALL INIT$LOAD$TABLE;
CALL BUILD;
CALL MOVE(.INTERP$FCB,FCB,33);
FCB$BYTE$A(32) = 0;
IF OPEN(FCB) = 255 THEN
    DO;
        CALL CRLF;
        CALL PRINT(.('CINTERP.COM NOT FOUND   $'));
        CALL REBOOT;
    END;
CALL MOVE(READER$LOCATION, 80H, 80H);
CALL MOVE(.HI$OFFSET,0FCH,4);
ADDR = 80H;
CALL ADDR; /* BRANCH TO 80H */
END;
```

COMPUTER LISTING FOR MODULE INTRDR NPS MICRO-COBOL

```
$ TITLE('NPS MICRO-COBOL COMPILER INTRDR') PAGEWIDTH(80)
     PAGELENGTH(60)
INTRDR: DO;

             /*     COBOL COMPILER - INTRDR     */

             /*     NORMALLY LOCATED AT 80 H     */

          /*     GLOBAL DECLARATIONS AND LITERALS     */

    /* THIS PROGRAM IS CALLED BY THE BUILD PROGRAM AFTER
    CINTERP.COM HAS BEEN OPENED, AND READS THE CODE INTO MEMORY
    */

DECLARE
    LIT     LITERALLY     'LITERALLY',
    DCL     LIT           'DECLARE',
    I       ADDRESS       INITIAL (0080H),
    INTERP  ADDRESS       INITIAL(100H),
    PROC    LIT           'PROCEDURE',
    START   LIT           '100H';

MON1:PROC(F,A) EXTERNAL;
    DCL F BYTE, A ADDRESS;
END MON1;

MON2: PROC(F,A) BYTE EXTERNAL;
    DCL F BYTE, A ADDRESS;
END MON2;

    DO WHILE 1;
        CALL MON1 (26,(I := I + 0080H)); /* SET DMA ADDRESS */
        IF MON2 (20,5CH) <> 0 THEN
            CALL INTERP;
    END;
END;
```

```
$ TITLE('NPS MICRO-COBOL COMPILER DECODE') PAGEWIDTH(80)
  PAGELENGTH(60)
DECODE: DO;

            /*    COBOL COMPILER - DECODE    */

            /*    NORMALLY LOCATED AT 103H    */

         /*    GLOBAL DECLARATIONS AND LITERALS    */

    /* THIS PROGRAM TAKES THE CODE OUTPUT FROM THE COBOL
    COMPILER AND CONVERTS IT INTO A READABLE OUTPUT TO
    FACILITATE DEBUGGING */

DECLARE DCL             LITERALLY       'DECLARE',
        LIT             LITERALLY       'LITERALLY',
        ADDR            ADDRESS         INITIAL (100H),
        BUFF$END        LIT             '0FFE',
        BYTE$COUNT      ADDRESS         INITIAL(0),
        BYTE$HI         BYTE,
        BYTE$LOW        BYTE,
        CHAR            BASED ADDR      BYTE,
        C$ADDR          BASED ADDR      ADDRESS,
        FCB             ADDRESS         INITIAL (5CH),
        FCB$BYTE        BASED FCB (1)   BYTE,
        FILE$TYPE(*)    BYTE            DATA ('CIN'),
        I               BYTE,
        PROC            LIT             'PROCEDURE';

MON1: PROC (F,A) EXTERNAL;
    DCL F BYTE, A ADDRESS;
END MON1;


MON2: PROC (F,A) BYTE EXTERNAL;
    DCL F BYTE, A ADDRESS;
END MON2;


BOOT: PROC EXTERNAL;
END BOOT;


PRINT$CHAR: PROC(CHAR);
    DCL CHAR BYTE;
    CALL MON1(2,CHAR);
END PRINT$CHAR;
```

```
CRLF: PROC;
    CALL PRINT$CHAR(13);
    CALL PRINT$CHAR(10);
END CRLF;


P: PROC(ADD1);
    DCL ADD1 ADDRESS, C BASED ADD1 (1) BYTE;
    CALL CRLF;
    DO I = 0 TO 2;
        CALL PRINT$CHAR(C(I));
    END;
    CALL PRINT$CHAR(' ');
END P;

GET$CHAR: PROC BYTE;
    IF (ADDR := ADDR + 1) > BUFF$END THEN
        DO;
            IF MON2(20,FCB) <> 0 THEN
                DO;
                    CALL P(.('END'));
                    CALL BOOT;
                END;
            ADDR = 80H;
        END;
    RETURN CHAR;
END GET$CHAR;


D$CHAR: PROC (OUTPUT$BYTE);
    DCL OUTPUT$BYTE BYTE;
    IF OUTPUT$BYTE < 10 THEN
        CALL PRINT$CHAR(OUTPUT$BYTE + 30H);
    ELSE
        CALL PRINT$CHAR(OUTPUT$BYTE + 37H);
END D$CHAR;


D: PROC (COUNT);
    DCL(COUNT,J) ADDRESS;
    DO J=1 TO COUNT;
        CALL D$CHAR(SHR(GET$CHAR,4));
        CALL D$CHAR(CHAR AND 0FH);
        CALL PRINT$CHAR(' ');
    END;
END D;


PRINT$REST: PROC;
```

```
DCL
F2    LIT  '9',
F3    LIT  '10',
F4    LIT  '22',
F5    LIT  '26',
F6    LIT  '34',
F7    LIT  '41',
F8    LIT  '51',
F9    LIT  '51',
F10   LIT  '56',
F11   LIT  '62',
F12   LIT  '63',
SBR   LIT  '64',
F13   LIT  '63',
GDP   LIT  '65',
PAR   LIT  '66',
INT   LIT  '67',
BST   LIT  '68',
TER   LIT  '69',
SCD   LIT  '70';

IF CHAR < F2  THEN RETURN;
IF CHAR < F3  THEN DO; CALL D(1);  RETURN; END;
IF CHAR < F4  THEN DO; CALL D(2);  RETURN; END;
IF CHAR < F5  THEN DO; CALL D(3);  RETURN; END;
IF CHAR < F6  THEN DO; CALL D(4);  RETURN; END;
IF CHAR < F7  THEN DO; CALL D(5);  RETURN; END;
IF CHAR < F8  THEN DO; CALL D(6);  RETURN; END;
IF CHAR < F9  THEN DO; CALL D(7);  RETURN; END;
IF CHAR < F10 THEN DO; CALL D(8);  RETURN; END;
IF CHAR < F11 THEN DO; CALL D(9);  RETURN; END;
IF CHAR < F12 THEN DO; CALL D(10); RETURN; END;
IF CHAR < F13 THEN DO; CALL D(11); RETURN; END;
IF CHAR < SBR THEN DO; CALL D(12); RETURN; END;
IF CHAR = SBR THEN DO; CALL D(8); RETURN; END;
IF CHAR = GDP THEN
    DO;
        CALL D(1);
        CALL D(SHL(CHAR.1) + 3);
        RETURN;
    END;
IF CHAR = PAR THEN
    DO;
        CALL D(1);
        CALL D(SHL(CHAR.1) + 1);
        RETURN;
    END;
IF CHAR = INT THEN
    DO;
        BYTE$COUNT = 0;
        CALL D(3);
```

```
            BYTE$LOW = CHAR;
            CALL D(1);
            BYTE$HI = CHAR;
            BYTE$COUNT = BYTE$HI;
            BYTE$COUNT = SHL(BYTE$COUNT,8) + BYTF$LOW;
            CALL D(BYTE$COUNT);
            RETURN;
        END;
    IF CHAR = BST THEN
        DO;
            CALL D(4);
            RETURN;
        END;
    IF CHAR = TER THEN
        DO;
            CALL D(2);
            CALL P(.('END'));
            CALL BOOT;
        END;
    IF CHAR = SCD THEN
        DO;
            CALL D(2);
            RETURN;
        END;
    CALL P(.('XXX'));
END PRINT$REST;


    /* PROGRAM EXECUTION STARTS HERE */

FCB$BYTE(32), FCB$BYTE(0) = 0;
DO I=0 TO 2;
    FCB$BYTE(I+9)=FILE$TYPE(I);
END;

IF MON2(15,FCB)=255 THEN DO; CALL P(.('ZZZ'));
                             CALL BOOT; END;

DO WHILE 1;
    IF GET$CHAR <= 70 THEN DO CASE CHAR;
    ;    /* CASE 0 NOT USED */
        CALL P(.('ADD'));
        CALL P(.('SUB'));
        CALL P(.('MUL'));
        CALL P(.('DIV'));
        CALL P(.('NEG'));
        CALL P(.('STP'));
        CALL P(.('STI'));
        CALL P(.('EXT'));
        CALL P(.('RND'));
        CALL P(.('RET'));
```

```
CALL P(.('CLS'));
CALL P(.('SER'));
CALL P(.('BRN'));
CALL P(.('OPN'));
CALL P(.('OP1'));
CALL P(.('OP2'));
CALL P(.('RGT'));
CALL P(.('RLT'));
CALL P(.('REQ'));
CALL P(.('INV'));
CALL P(.('EOR'));
CALL P(.('PAG'));
CALL P(.('ACC'));
CALL P(.('STD'));
CALL P(.('LDI'));
CALL P(.('DIS'));
CALL P(.('DEC'));
CALL P(.('STO'));
CALL P(.('ST1'));
CALL P(.('ST2'));
CALL P(.('ST3'));
CALL P(.('ST4'));
CALL P(.('ST5'));
CALL P(.('LOD'));
CALL P(.('LD1'));
CALL P(.('LD2'));
CALL P(.('LD3'));
CALL P(.('LD4'));
CALL P(.('LD5'));
CALL P(.('LD6'));
CALL P(.('PER'));
CALL P(.('CNU'));
CALL P(.('CNS'));
CALL P(.('CAL'));
CALL P(.('RWS'));
CALL P(.('DLS'));
CALL P(.('RDF'));
CALL P(.('WTF'));
CALL P(.('RVL'));
CALL P(.('WVL'));
CALL P(.('SCR'));
CALL P(.('SGT'));
CALL P(.('SLT'));
CALL P(.('SEQ'));
CALL P(.('MOV'));
CALL P(.('RRS'));
CALL P(.('WRS'));
CALL P(.('RRR'));
CALL P(.('WRR'));
CALL P(.('RWR'));
CALL P(.('DLR'));
```

```
        CALL P(.('MED'));
        CALL P(.('MNE'));
        CALL P(.('SBR'));
        CALL P(.('GDP'));
        CALL P(.('PAR'));
        CALL P(.('INT'));
        CALL P(.('BST'));
        CALL P(.('TFR'));
        CALL P(.('SCD'));
    END; /* OF CASE STATEMENT */
    CALL PRINT$REST;
END; /* END OF DO WHILE */
END;
```

GRAMMER FOR PART ONE NPS MICRO-COBOL

```
OPTIONS (BNF TABLES LALR AINPUT EXTRAT NOGPOST COMPACT)

 1   <PROGRAM> ::= <ID-DIV> <E-DIV> <D-DIV> PROCEDURE
 2   <ID-DIV> ::= IDENTIFICATION DIVISION . PROGRAM-ID .
 2                    <COMMENT> . <ID-LIST>
 3   <ID-LIST> ::= <AUTH> <INS> <DATE> <SEC>
 4   <AUTH> ::= AUTHOR . <COMMENT> .
 5              <EMPTY>
 6   <INS> ::= INSTALLATION . <COMMENT> .
 7             <EMPTY>
 8   <DATE> ::= DATE-WRITTEN . <COMMENT> .
 9              <EMPTY>
10  <SEC> ::= SECURITY . <COMMENT> .
11            <EMPTY>
12   <COMMENT> ::= <INPUT>
13                 <COMMENT> <INPUT>
14   <E-DIV> ::= ENVIRONMENT DIVISION . CONFIGURATION
14                  SECTION . <SRC-OBJ> <I-O>
15               <EMPTY>
16   <SRC-OBJ> ::= SOURCE-COMPUTER . <COMMENT> <DEBUG> .
16                   OBJECT-COMPUTER . <COMMENT> .
17   <DEBUG> ::= DEBUGGING MODE
18               <EMPTY>
19   <I-O> ::= INPUT-OUTPUT SECTION . FILE-CONTROL .
19                <FILE-CONTROL-LIST> <IC>
20             <EMPTY>
21   <FILE-CONTROL-LIST> ::= <FILE-CONTROL-ENTRY>
22                           <FILE-CONTROL-LIST>
22                               <FILE-CONTROL-ENTRY>
23   <FILE-CONTROL-ENTRY> ::= SELECT <ID> <ATTRIBUTE-LIST> .
24   <ATTRIBUTE-LIST> ::= <ONE-ATTRIB>
25                        <ATTRIBUTE-LIST> <ONE-ATTRIB>
26   <ONE-ATTRIB> ::= ORGANIZATION <ORG-TYPE>
27                    ACCESS <ACC-TYPE> <RELATIVE>
28                    ASSIGN <INPUT>
29   <ORG-TYPE> ::= SEQUENTIAL
30                  RELATIVE
31                  INDEXED
32   <ACC-TYPE> ::= SEQUENTIAL
33                  RANDOM
34   <RELATIVE> ::= RELATIVE <ID>
35                  <EMPTY>
36   <IC> ::= I-O-CONTROL . <SAME-LIST>
37            <EMPTY>
38   <SAME-LIST> ::= <SAME-ELEMENT>
39                   <SAME-LIST> <SAME-ELEMENT>
40   <SAME-ELEMENT> ::= SAME <ID-STRING> .
```

```
41   <ID-STRING> ::= <ID>
42                   <ID-STRING> <ID>
43   <D-DIV> ::= DATA DIVISION . <FILE-SECTION> <WORK>
43                   <LINK>
44   <FILE-SECTION> ::= FILE SECTION . <FILE-LIST>
45                      <EMPTY>
46   <FILE-LIST> ::= <FILES>
47                   <FILE-LIST> <FILES>
48   <FILES> ::= FD <ID> <FILE-CONTROL> .
48                   <RECORD-DESCRIPTION>
49   <FILE-CONTROL> ::= <FILE-LST>
50                      <EMPTY>
51   <FILE-LST> ::= <FILE-ELEMENT>
52                  <FILE-LST> <FILE-ELEMENT>
53   <FILE-ELEMENT> ::= BLOCK <INTEGER> RECORDS
54                      RECORD <REC-COUNT>
55                      LABEL RECORDS STANDARD
56                      LABEL RECORDS OMITTED
57                      VALUE OF <ID-STRING>
58  <REC-COUNT> ::= <INTEGER>
59                  <INTEGER> TO <INTEGER>
60  <WORK> ::= WORKING-STORAGE SECTION .
60                <RECORD-DESCRIPTION>
61              <EMPTY>
62  <LINK> ::= LINKAGE SECTION . <RECORD-DESCRIPTION>
63              <EMPTY>
64  <RECORD-DESCRIPTION> ::= <LEVEL-ENTRY>
65                           <RECORD-DESCRIPTION>
65                              <LEVEL-ENTRY>
66   <LEVEL-ENTRY> ::= <INTEGER> <DATA-ID> <REDEFINES>
66                        <DATA-TYPE> .
67   <DATA-ID> ::= <ID>
68                 FILLER
69   <REDEFINES> ::= REDEFINES <ID>
70                   <EMPTY>
71   <DATA-TYPE> ::= <PROP-LIST>
72                   <EMPTY>
73   <PROP-LIST> ::= <DATA-ELEMENT>
74             <PROP-LIST> <DATA-ELEMENT>
75   <DATA-ELEMENT> ::= PIC <INPUT>
76                      USAGE COMP
77                      USAGE COMP-3
78                      USAGE COMPUTATIONAL
79                      USAGE DISPLAY
80                      SIGN LEADING <SEPARATE>
81                      SIGN TRAILING <SEPARATE>
82                      OCCURS <INTEGER> INDEXED <ID>
83                      OCCURS <INTEGER>
84                      SYNC <DIRECTION>
85                      VALUE <LITERAL>
86   <DIRECTION> ::= LEFT
```

```
87                      RIGHT
88                      <EMPTY>
89    <SEPARATE> ::= SEPARATE
90                     <EMPTY>
91    <LITERAL> ::= <INPUT>
92                  <LIT>
93                  ZERO
94                  SPACE
95                  QUOTE
96    <INTEGER> ::= <INPUT>
97    <ID> ::= <INPUT>
```

Note that the options list contains the item NOGPOST. This elimenats the hoal symbol "_|_" from being added to the grammer of part one. In part two the hoal symbol is used as an end of file symbol (EOF).

GRAMMER FOR PART TWO NPS MICRO-COBOL

```
OPTIONS (BNF TABLES LALR AINPUT EXTRAT COMPACT)

 1   <P-DIV> ::= PROCEDURE DIVISION <USING> . <PROC-BODY>
 2   <USING> ::= USING <ID-STRING>
 3               <EMPTY>
 4   <ID-STRING> ::= <ID>
 5                   <ID-STRING> <ID>
 6   <PROC-BODY> ::= <PARAGRAPH>
 7                  <PROC-BODY> <PARAGRAPE>
 8   <PARAGRAPH> ::= <ID> .
 9                   <ID> . <SENTENCE-LIST>
10                   <ID> SECTION .
11   <SENTENCE-LIST> ::= <SENTENCE> .
12                       <SENTENCE-LIST> <SENTENCE> .
13   <SENTENCE> ::= <IMPERATIVE>
14                 <CONDITIONAL>
15                 ENTER <ID> <OPT-ID>
16   <IMPERATIVE> ::= ACCEPT <SUBID>
17                    <ARITHMETIC>
18                    CALL <CALL-LIT> <USING>
19                    CLOSE <CLOSE-LST>
20                    <FILE-ACT>
21                    DISPLAY <DISPLAY-LST>
22                    DISPLAY <DISPLAY-LST> WITH NO
22                        ADVANCING
23                    EXIT <PROGRAM-ID>
24                    GO <ID>
25                    GO <ID-STRING> DEPENDING <ID>
26                    MOVE <LIT/ID> TO <SUBID>
27                    OPEN <ACT-LST>
28                    PERFORM <ID> <THRU> <FINISH>
29                    STOP <TERMINATE>
30   <CLOSE-LST> ::= <ID>
31                  <CLOSE-LST> <ID>
32   <DISPLAY-LST> ::= <LIT/ID>
33                     <DISPLAY-LST> <LIT/ID>
34   <ACT-LST> ::= <TYPE-ACTION> <OPEN-LST>
35                <ACT-LST> <TYPE-ACTION> <OPEN-LST>
36   <OPEN-LST> ::= <ID>
37                  <OPEN-LST> <ID>
38   <FINISH> ::= <L/ID> TIMES
39                <STOPCONDITION>
40                <VARYING> <ITERATION> <STOPCONDITION>
41                <EMPTY>
42   <STOPCONDITION> ::= UNTIL <CONDITION>
43   <VARYING> ::= VARYING <SUBID>
44   <ITERATION> ::= <FROM> <BY>
```

```
45   <FROM> ::= FROM <L/ID>
46   <BY> ::= BY <L/ID>
47   <CONDITIONAL> ::= <ARITHMETIC> <SIZE-ERROR>
47                         <IMPERATIVE>
48                    <FILE-ACT> <INVALID> <IMPERATIVE>
49                    <READ-ID> <SPECIAL> <IMPERATIVE>
50                    <IF-NONTERMINAL> <CONDITION>
50                         <IF-LST> <ELSE> <IF-LST> END-IF
51                    <IF-NONTERMINAL> <CONDITION>
51                         <IF-LST> END-IF
52   <IF-LST> ::= <STMT-LST>
53                NEXT SENTENCE
54   <ELSE> ::= ELSE
55   <ARITHMETIC> ::= ADD <ADD-LST> TO <SUBID> <ROUND>
56                    ADD <ADD-LST>GIVING <SUBID> <ROUND>
57                    DIVIDE <L/ID> INTO <SUBID> <ROUND>
58                    DIVIDE <L/ID> BY <SUBID> GIVING
58                        <SUBID> <ROUND>
59                    DIVIDE <L/ID> INTO <SUBID> GIVING
59                        <SUBID> <ROUND>
60                    MULTIPLY <L/ID> BY <SUBID> <ROUND>
61                    MULTIPLY <L/ID> BY <SUBID> GIVING
61                        <SUBID> <ROUND>
62                    SUBTRACT <SUB-LST> FROM <SUBID>
62                        <ROUND>
63                    SUBTRACT <SUB-LST> GIVING <SUBID>
63                        <ROUND>
64                    COMPUTE <SUBID> = <ARITH-EXP>
65   <ADD-LST> ::= <L/ID>
66                 <ADD-LST> <L/ID>
67   <SUB-LST> ::= <L/ID>
68                 <SUB-LST> <L/ID>
69   <ARITH-EXP> ::= <TERM>
70                   <ARITH-EXP> + <TERM>
71                   <ARITH-EXP> - <TERM>
72                   + <TERM>
73                   - <TERM>
74   <TERM> ::= <PRIMARY>
75              <TERM> * <PRIMARY>
76              <TERM> / <PRIMARY>
77   <PRIMARY> ::= <PRIM-ELEM>
78                 <PRIMARY> ** <PRIM-ELEM>
79   <PRIM-ELEM> ::= <L/ID>
80                   ( <ARITH-EXP> )
81   <FILE-ACT> ::= DELETE <ID>
82                  REWRITE <ID>
83                  WRITE <ID> <SPECIAL-ACT>
84   <CONDITION> ::= <BTERM>
85                   <CONDITION> OR <BTERM>
86   <BTERM> ::= <BPRIM>
87               <BTERM> AND <BPRIM>
```

```
 88   <BPRIM> ::= <LIT/ID>
 89               <LIT/ID> <NOT> <COND-TYPE>
 90               ( <BTERM> )
 91   <COND-TYPE> ::= NUMERIC
 92                   ALPHABETIC
 93                   <COMPARE> <LIT/ID>
 94   <NOT> ::= NOT
 95             <EMPTY>
 96   <COMPARE> ::= GREATER
 97                 LESS
 98                 EQUAL
 99                 >
100                 <
101                 =
102   <ROUND> ::= ROUNDED
103               <EMPTY>
104   <TERMINATE> ::= <LITERAL>
105                   RUN
106   <SPECIAL> ::= <INVALID>
107                 END
108   <OPT-ID> ::= <SUBID>
109                <EMPTY>
110   <STMT-LST> ::= <IMPERATIVE>
111                  <STMT-LST> <IMPERATIVE>
112                  <CONDITIONAL>
113                  <STMT-LST> <CONDITIONAL>
114   <THRU> ::= THRU <ID>
115              <EMPTY>
116   <INVALID> ::= INVALID
117   <SIZE-ERROR> ::= SIZE ERROR
118   <SPECIAL-ACT> ::= <WHEN> ADVANCING <HOW-MANY>
119                     <EMPTY>
120   <WHEN> ::= BEFORE
121              AFTER
122   <HOW-MANY> ::= <INTEGER>
123                  PAGE
124   <TYPE-ACTION> ::= INPUT
125                     OUTPUT
126                     I-O
127   <SUBID> ::= <SUBSCRIPT>
128               <ID>
129   <INTEGER> ::= <INPUT>
130   <ID> ::= <INPUT>
131   <L/ID> ::= <INPUT>
132              <SUBSCRIPT>
133              ZERO
134   <SUBSCRIPT> ::= <ID> ( <SUBSCRIPT-LST> )
135   <SUBSCRIPT-LST> ::= <INPUT>
136                       <SUBSCRIPT-LST> , <INPUT>
137   <CALL-LIT> ::= <LIT>
138   <NN-LIT> ::= <LIT>
```

```
139                 SPACE
140                 QUOTE
141   <LITERAL> ::= <NN-LIT>
142                  <INPUT>
143                  ZERO
144   <LIT/ID> ::= <L/ID>
145                <NN-LIT>
146   <PROGRAM-ID> ::= <ID>
147                    <EMPTY>
148   <READ-ID> ::= READ <ID>
149   <IF-NONTERMINAL> ::= IF
```

Note that the options list does not contain the item NOGPOST. This causes a goal symbol "_!_" to be added to the grammer at the end of production one. This symbol is used as the end of file symbol (EOF). Part one uses the optional NOGPOST to surpress the generation of the goal symbol since an EOF is not wanted at the end of part one.

LIST OF REFERENCES

1. Aho, A. V. and Ullman, J. D., Principles of Compiler Design, Addison-Wesley, 1977.

2. Cagle, Carol, paper presented as a term project for CS3113, Naval Postgraduate School, Monterey, California, Fall Term, 1979.

3. Craig, A. S., MICRO-COBOL An Implementation of Navy Standard HYPO-COBOL for a Micro-processor based Computer System, Master's Thesis, Naval Postgraduate School, Monterey, California, 1977.

4. Department of the Navy Automated Data Processing Equipment Selection Office, HYPO-COBOL Validation System (HCCVS), April 1975.

5. Digital Research, An Introduction to CP/M Features and Facilites, 1976.

6. Digital Research, CP/M Interface Guide, 1976.

7. Digital Research, Symbolic Instruction Debugger User's Guide, 1978.

8. Digital Research, CP/M Dynamic Debugging Tool (DDT) User's Guide, 1976.

9. Farlee J. and Rice, M., NPS MICRO-COBOL an Implementation of Navy Standard HYPO-COBOL for a Microprocesor-Based Computer System, Master's Thesis, Naval Postgraduate School, Monterey, California, 1979.

10. Intel Corporation, PL/M-80 Programming Manual, 1976.

11. Intel Corporation, ISIS-II User's Guide, 1976.

12. Intel Corporation, ISIS-II PL/M-80 Compiler Operator's Manual, 1976.

13. Intel Corporation, 8008 and 8080 PL/M Programming Manual, 1975.

14. Kiefer, R. and Perry, C., MICRO-COBOL Validation System, paper presented as a term project for CS3020, Naval Postgraduate School, Monterey, California, Fall Term, 1978.

15. Loskot, Doug, paper presented as a term project for CS3400,

Naval Postgraduate School, Monterey, California, Winter Term, 1979.

16. McCracken, D. M., A Simplified Guide to Structured COBOL Programming, Wiley, 1976.

17. Myers, G. J., Software Reliability--Principles and Practices, Wiley, 1976.

18. Mylet, P. R., MICRO-COBOL A Subset of Navy Standard HYPO-COBOL for Micro-computers, Master's Thesis, Naval Postgraduate School, Monterey, California, 1978.

19. Stowers, D. and Hartel, R., paper presented as a term project for CS3113, Naval Postgraduate School, Monterey, California, Summer Term, 1979.

20. University of Toronto Computer Systems Research Group Technical Report CSRG-2, An Efficient LALR Parser Generator by W. R. Lalonge, April 1971.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22314	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93940	2
3.	Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93940	3
4.	Asst. Professor L. A. Cox Jr., Code 52Cl Department of Computer Science Naval Postgraduate School Monterey, California 93940	1
5.	Lt. M. S. Moranville, Code 52Mi Naval Electronics Systems Engineering Center San Diego, California 92101	2
6.	ADPE Selection Office Department of the Navy Washington, D. C. 20376	1
7.	Lt. Hal R. Powell 1295 Heatherstone Wy Sunnyvale, California 94087	2
8.	Gordon Eubanks 923 Kapoho Pl. Hawaii Kai, Hawaii 96825	1
9.	Lcdr Robert R. Stilwell Code 52SB Naval Postgraduate School Monterey, California 93940	1
10.	Carol Cagle 1060 Halsey Dr. Monterey, California 93940	1

Thesis
P7633
c.1

189085

Powell
NPS MICRO-COBOL an
implementation of a
subset of ANSI-COBOL
for a microcomputer
system.

Thesis
P7633
c.1

189085

Powell
NPS MICRO-COBOL an
implementation of a
subset of ANSI-COBOL
for a microcomputer
system.

COBOL :
3 2768 001 93148 8
DUDLEY KNOX LIBRARY